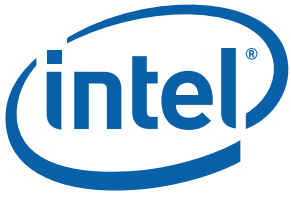


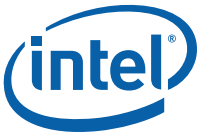
Intel® Math Kernel Library for Linux*

User's Guide

October 2007

Document Number: 314774-005US

World Wide Web: http://developer.intel.com



Version	Version Information	Date
-001	Original issue. Documents Intel® Math Kernel Library (Intel® MKL) 9.0 gold release.	September 2006
-002	Documents Intel® MKL 9.1 beta release. "Getting Started", "LINPACK and MP LINPACK Benchmarks" chapters and "Support for Third-Party and Removed Interfaces" appendix added. Existing chapters extended. Document restructured. List of examples added.	January 2007
-003	Documents Intel® MKL 9.1 gold release. Existing chapters extended. Document restructured. More aspects of ILP64 interface discussed. Section "Configuring Eclipse CDT to Link with Intel MKL" added to chapter 3. Cluster content is organized into one separate chapter 9 "Working with Intel® Math Kernel Library Cluster Software" and restructured, appropriate links added.	June 2007
-004	Documents Intel® MKL 10.0 Beta release. Layered design model has been described in chapter 3 and the content of the entire book adjusted to the model. Automation of setting environment variables at startup has been described in chapter 4. New Intel MKL threading controls have been described in chapter 6. The User's Guide for Intel MKL merged with the one for Intel MKL Cluster Edition to reflect consolidation of the respective products.	September 2007
-005	Documents Intel® MKL 10.0 Gold release. Configuring of Eclipse CDT 4.0 to link with Intel MKL has been described in chapter 3. Intel® Compatibility OpenMP* run-time compiler library (`libiomp`) has been described.	October 2007

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2006 - 2007, Intel Corporation. All rights reserved.

Contents

Chapter 1 Overview

Technical Support 1-1
About This Document 1-1
Purpose 1-2
Audience 1-2
Document Organization 1-2
Notational Conventions 1-3

Chapter 2 Getting Started

Checking Your Installation 2-1
Obtaining Version Information 2-2
Compiler Support 2-2
Before You Begin Using Intel MKL 2-2

Chapter 3 Intel® Math Kernel Library Structure

High-level Directory Structure 3-1
Layered Model Concept 3-2
Layers 3-3
Sequential Version of the Library 3-4
Support for ILP64 Programming 3-5
Intel MKL Versions 3-11
Directory Structure in Detail 3-11
Dummy Libraries 3-19
Contents of the Documentation Directory 3-19

Chapter 4 Configuring Your Development Environment

Setting Environment Variables 4-1
Automating the Process 4-1
Configuring Eclipse CDT to Link with Intel MKL 4-2
Configuring Eclipse CDT 4.0 4-2
Configuring Eclipse CDT 3.x 4-3
Customizing the Library Using the Configuration File 4-4

Chapter 5 Linking Your Application with Intel® Math Kernel Library

Selecting Between Linkage Models 5-1
Static Linking 5-1
Dynamic Linking 5-2
Making the Choice 5-2
Intel MKL-specific Linking Recommendations 5-3
Link Command Syntax 5-3
Selecting Libraries to Link 5-6
Linking with Threading Libraries 5-7
More Linking Examples 5-8
Notes on Linking 5-10
Building Custom Shared Objects 5-11
Intel MKL Custom Shared Object Builder 5-11
Specifying Makefile Parameters 5-11
Specifying List of Functions 5-12

Chapter 6 Managing Performance and Memory

Using Intel MKL Parallelism 6-1
Techniques to Set the Number of Threads 6-2
Avoiding Conflicts in the Execution Environment 6-3
Setting the Number of Threads Using OpenMP Environment Variable ... 6-4
Changing the Number of Threads at Run Time 6-4
Using Additional Threading Control 6-7
Tips and Techniques to Improve Performance 6-12
Coding Techniques 6-12
Hardware Configuration Tips 6-13

Managing Multi-core Performance 6-14
Operating on Denormals..... 6-15
FFT Optimized Radices 6-15
Using Intel MKL Memory Management..... 6-15
Redefining Memory Functions..... 6-16

Chapter 7 Language-specific Usage Options

Using Language-Specific Interfaces with Intel MKL..... 7-1
Mixed-language programming with Intel MKL 7-4
Calling LAPACK, BLAS, and CBLAS Routines from C Language Environments 7-4
Calling BLAS Functions That Return the Complex Values in C/C++ Code 7-6
Invoking Intel MKL Functions from Java Applications 7-9

Chapter 8 Coding Tips

Aligning Data for Numerical Stability 8-1

Chapter 9 Working with Intel® Math Kernel Library Cluster Software

Linking with ScaLAPACK and Cluster FFTs 9-1
Setting the Number of Threads 9-2
Using Shared Libraries 9-3
ScaLAPACK Tests..... 9-3
Examples for Linking with ScaLAPACK and Cluster FFT 9-3
Examples for C Module..... 9-3
Examples for Fortran Module..... 9-4

Chapter 10 LINPACK and MP LINPACK Benchmarks

Intel® Optimized LINPACK Benchmark for Linux* 10-1
Contents 10-1
Running the Software 10-2
Known Limitations 10-3
Intel® Optimized MP LINPACK Benchmark for Clusters 10-4
Contents 10-4
Building MP LINPACK 10-6
New Features..... 10-6

Benchmarking a Cluster 10-6

Appendix A Intel® Math Kernel Library Language Interfaces Support

Appendix B Support for Third-Party Interfaces

GMP* Functions B-1
FFTW Interface Support B-1

Index

List of Tables

Table 1-1 Notational conventions 1-4
Table 2-1 What you need to know before you get started 2-2
Table 3-1 High-level directory structure 3-1
Table 3-2 Intel MKL ILP64 concept 3-7
Table 3-3 Compiler options for ILP64 interface 3-8
Table 3-4 Integer types 3-8
Table 3-5 Intel MKL include files 3-9
Table 3-6 ILP64 support in Intel MKL 3-10
Table 3-7 Detailed directory structure 3-11
Table 3-8 Contents of the doc directory 3-19
Table 5-1 Quick comparison of Intel MKL linkage models 5-2
Table 5-2 Selecting the Threading and RTL layer 5-8
Table 6-1 How to avoid conflicts in the execution environment for your threading model 6-3
Table 6-2 Intel MKL environment variables for threading controls 6-8
Table 6-3 Interpretation of MKL_DOMAIN_NUM_THREADS values 6-10
Table 7-1 Interface libraries and modules 7-1
Table 10-1 Contents of the LINPACK Benchmark 10-2
Table 10-2 Contents of the MP LINPACK Benchmark 10-5

List of Examples

Example 4-1 Intel MKL configuration file 4-4
Example 4-2 Redefining library names using the configuration file 4-5
Example 6-1 Changing the number of processors for threading 6-5
Example 6-2 Setting the number of threads to one 6-8
Example 6-3 Setting an affinity mask by operating system means using Intel® compiler 6-14
Example 6-4 Redefining memory functions 6-17
Example 7-1 Calling a complex BLAS Level 1 function from C 7-7
Example 7-2 Calling a complex BLAS Level 1 function from C++ 7-8
Example 7-3 Using CBLAS interface instead of calling BLAS directly from C 7-9
Example 8-1 Aligning addresses at 16-byte boundaries 8-2

Overview

1

Intel® Math Kernel Library (Intel® MKL) offers highly optimized, thread-safe math routines for science, engineering, and financial applications that require maximum performance.

Technical Support

Intel provides a support web site, which contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more. Visit the Intel® MKL support website at http://www.intel.com/software/products/support/ .

About This Document

To succeed in developing applications with Intel MKL, information of two kinds is basically required. *Reference* information covers routine functionality, parameter descriptions, interfaces and calling syntax as well as return values. To get this information, see *Intel MKL Reference Manual* first. However, a lot of questions not answered in the Reference Manual arise when you try to call Intel MKL routines from your applications. For example, you need to know how the library is organized, how to configure Intel MKL for your particular platform and problems you are solving, how to compile and link your applications with Intel MKL. You also need understanding of how to obtain best performance, take advantage of Intel MKL threading and memory management. Other questions may deal with specifics of routine calls, for example, passing parameters in different programming languages or coding inter-language routine calls. You may be interested in the ways of estimating and improving computation accuracy. These and similar issues make up Intel MKL *usage information*.

This document focuses on the usage information needed to call Intel MKL routines from user's applications running on Linux*. Linux usage of Intel MKL has its particular features, which are described in this guide, along with those that do not depend upon a particular OS.

This guide contains usage information for Intel MKL routines and functions comprised in the function domains listed in Table A-1 (in Appendix A).

It is assumed that you use this document after the installation of the Intel MKL is complete on your machine. If you have not installed the product yet, use the *Intel MKL Installation Guide* (file `Install.txt`) for assistance.

The user's guide should be used in conjunction with the latest version of the *Intel® Math Kernel Library for Linux* Release Notes* document to reference how to use the library in your application.

Purpose

Intel® Math Kernel Library for Linux* User's Guide is intended to assist in mastering the usage of the Intel MKL on Linux. In particular, it

- Helps you start using the library by describing the steps you need to perform after the installation of the product
- Shows you how to configure the library and your development environment to use the library
- Acquaints you with the library structure
- Explains in detail how to link your application to the library and provides simple usage scenarios
- Describes various details of how to code, compile, and run your application with Intel MKL for Linux.

Audience

The guide is intended for Linux programmers whose software development experience may vary from beginner to advanced.

Document Organization

The document contains the following chapters and appendices.

Chapter 1	Overview. Introduces the concept of the Intel MKL usage information, describes the document's purpose and organization as well as explains notational conventions.
Chapter 2	Getting Started. Describes necessary steps and gives basic information needed to start using Intel MKL after its installation.

Chapter 3	Intel® Math Kernel Library Structure. Discusses the structure of the Intel MKL directory after installation at different levels of detail as well as the library versions and parts.
Chapter 4	Configuring Your Development Environment. Explains how to configure Intel MKL and your development environment for the use with the library.
Chapter 5	Linking Your Application with Intel® Math Kernel Library. Compares static and dynamic linking models; describes the general link line syntax to be used for linking with Intel MKL libraries; explains which libraries should be linked with your application for your particular platform and function domain; discusses how to build custom dynamic libraries.
Chapter 6	Managing Performance and Memory. Discusses Intel MKL threading; shows coding techniques and gives hardware configuration tips for improving performance of the library; explains features of the Intel MKL memory management and, in particular, shows how to replace memory functions that the library uses by default with your own ones.
Chapter 7	Language-specific Usage Options. Discusses mixed-language programming and the use of language-specific interfaces.
Chapter 8	Coding Tips. Presents coding tips that may be helpful to meet certain specific needs.
Chapter 9	Working with Intel® Math Kernel Library Cluster Software. Discusses usage of ScaLAPACK and Cluster FFTs mainly describing linking of your application with these function domains, including C- and Fortran-specific linking examples; gives information on the supported MPI.
Chapter 10	LINPACK and MP LINPACK Benchmarks. Describes the Intel® Optimized LINPACK Benchmark for Linux* and Intel® Optimized MP LINPACK Benchmark for Clusters.
Appendix A	Intel® Math Kernel Library Language Interfaces Support. Summarizes information on language interfaces that Intel MKL provides for each function domain.
Appendix B	Support for Third-Party Interfaces. Describes in brief certain interfaces that Intel MKL supports.

The document also includes an Index.

Notational Conventions

The document employs the following font conventions and symbols:

Table 1-1 Notational conventions

Italic	Italic is used for emphasis and also indicates document names in body text, for example: see *Intel MKL Reference Manual*
Monospace lowercase	Indicates filenames, directory names and pathnames, for example: `libmkl_core.a , /opt/intel/mkl/10.0.039`

Monospace lowercase mixed with uppercase	Indicates commands and command-line options, for example:
	`icc myprog.c -L$MKLPATH -I$MKLINCLUDE -lmkl -lguide -lpthread ;`
	C/C++ code fragments, for example:
	`a = new double [SIZE*SIZE];`

UPPERCASE MONOSPACE	Indicates system variables, for example, `$MKLPATH`
Monospace italic	Indicates a parameter in discussions: routine parameters, for example, *lda*; makefile parameters, for example, *functions_list*; etc. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value, for example, *<mkl directory>*. Substitute one of these items for the placeholder.
[items]	Square brackets indicate that the items enclosed in brackets are optional.
{ item | item }	Braces indicate that only one of the items listed between braces should be selected. A vertical bar (|) separates the items

Getting Started 2

This chapter helps you start using the Intel® Math Kernel Library (Intel® MKL) for Linux* by giving basic information you need to know and describing the necessary steps you need to perform after the installation of the product.

Checking Your Installation

Once you complete the installation of the Intel MKL, it is useful to perform a basic verification task that confirms proper installation and configuration of the library.

1. Check that the directory you chose for installation has been created. The default installation directory is `/opt/intel/mkl/10.0.xxx`, where `xxx` is the package number, for example, `/opt/intel/mkl/10.0.039`
2. Update build scripts so that they point to the *desired* version of Intel MKL if you choose to keep multiple versions installed on your computer. Note that you can have several versions of Intel MKL installed on your computer, but when installing, you are required to remove Beta versions of this software.
3. Check that the following six files are placed in the `tools/environment` directory:

   `mklvars32.sh`

   `mklvarsem64t.sh`

   `mklvars64.sh`

   `mklvars32.csh`

   `mklvarsem64t.csh`

   `mklvars64.csh`

   You can use these files to set environmental variables, such as `INCLUDE`, `LD_LIBRARY_PATH`, and `MANPATH`, in the current user shell.

Obtaining Version Information

Intel MKL provides a facility by which you can obtain information about the library (for example, the version number). Two methods are provided for extracting this information. First, you may extract a version string using the function `MKLGetVersionString`. Or, alternatively, you can use the `MKLGetVersion` function to obtain the `MKLVersion` structure which contains the version information. See the *Support Functions* chapter in the *Intel MKL Reference Manual* for the function descriptions and calling syntax. Example programs for extracting version information are provided in the `examples/versionquery` directory. A makefile is also provided to automatically build the examples and output summary files containing the version information for the current library.

Compiler Support

Intel supports Intel® MKL for use only with compilers identified in the *Release Notes*. However, the library has been successfully used with other compilers as well.

When using the CBLAS interface, the header file `mkl.h` will simplify program development, since it specifies enumerated values as well as prototypes for all the functions. The header determines if the program is being compiled with a C++ compiler and, if it is, the included file will be correct for use with C++ compilation.

Starting with Intel MKL 9.1, full support is provided for the GNU gfortran* compiler, which differs from the Intel® Fortran 9.1 compiler in calling conventions for functions that return complex data. Absoft Fortran compilers are supported as well.

Before You Begin Using Intel MKL

Before you get started using the Intel MKL, sorting out a few important basic concepts will greatly help you get off to a good start.

The table below summarizes some important things to think of before you start using Intel MKL.

Table 2-1 What you need to know before you get started

Target platform	Identify the architecture of your target machine: • IA-32 • Intel® 64 • IA-64 (Itanium® processor family) **Reason.** When linking your application with the Intel MKL libraries, the directory corresponding to your particular architecture should be included in the link command (see Selecting Libraries to Link)

Table 2-1 What you need to know before you get started (continued)

Mathematical problem	Identify all Intel MKL function domains that problems you are solving require: • BLAS • Sparse BLAS • LAPACK • ScaLAPACK • Sparse Solver routines • Vector Mathematical Library functions • Vector Statistical Library functions • Fourier Transform functions (FFT) • Cluster FFT • Interval Solver routines • Trigonometric Transform routines • Poisson, Laplace, and Helmholtz Solver routines • Optimization (Trust-Region) Solver routines **Reason.** The function domain you intend to use narrows the search in the *Reference Manual* for specific routines you need. Additionally, the link line that you use to link your application with Intel MKL cluster software depends on the function domains you intend to employ (see Working with Intel® Math Kernel Library Cluster Software). Note also that coding tips may depend on the function domain (see Tips and Techniques to Improve Performance).
Programming language	Though Intel MKL provides support for both Fortran and C/C++ programming, not all the function domains support a particular language environment, for example, C/C++ or Fortran90/95. Identify the language interfaces that your function domains support (see Intel® Math Kernel Library Language Interfaces Support). **Reason.** In case your function domain does not directly support the needed environment, you can use mixed-language programming. See Mixed-language programming with Intel MKL. See also Using Language-Specific Interfaces with Intel MKL for a list of language-specific interface libraries and modules and an example how to generate them.
Threading model	Select among the following options how you are going to thread your application: • Your application is already threaded • You may want to use the Intel® threading capability, that is, Legacy OpenMP* run-time library (`libguide`) or Compatibility OpenMP* run-time library (`libiomp`), or a threading capability provided by a third-party compiler • You do not want to thread your application. **Reason.** By default, OpenMP* sets the number of threads that Intel MKL uses. If you need a different number, you have to set it yourself using one of the available mechanisms. For more information, and especially, how to avoid conflicts in the threaded execution environment, see Using Intel MKL Parallelism. Additionally, the compiler that you use to thread your application determines which threading library you should link with your application (see Linking with Threading Libraries),

Table 2-1 What you need to know before you get started (continued)

Linking model	Decide which linking model is appropriate for linking your application with Intel MKL libraries: • Static • Dynamic **Reason.** For information on the benefits of each linking model, link command syntax and examples, link libraries as well as on other linking topics, like how to save disk space by creating a custom dynamic library, see Linking Your Application with Intel® Math Kernel Library.
MPI used	**Reason:** To link your application with ScaLAPACK and/or Cluster FFT, the libraries corresponding to your particular MPI should be included in the link line (see Working with Intel® Math Kernel Library Cluster Software).

Intel® Math Kernel Library Structure

3

The chapter discusses the structure of the Intel® Math Kernel Library (Intel® MKL) and, in particular, the structure of the Intel MKL directory after installation at different levels of detail as well as the library versions and parts.

Starting with version 10.0, Intel MKL employs the layered model (see Layered Model Concept for details), which is a drastic design change aimed to streamline the library structure, reduce its size, and add usage flexibility.

High-level Directory Structure

Table 3-1 shows a high-level directory structure of Intel MKL after installation.

Table 3-1 High-level directory structure

Directory	Comment
<mkl directory>	Main directory; by default "/opt/intel/mkl/10.0.xxx", where xxx is the Intel MKL package number, for example, "/opt/intel/mkl/10.0.039"
<mkl directory>/doc	Documentation directory
<mkl directory>/man/man3	Man pages for Intel MKL functions
<mkl directory>/examples	Source and data for examples
<mkl directory>/include	Contains INCLUDE files for both library routines and test and example programs
<mkl directory>/interfaces/blas95	Contains Fortran 95 wrappers for BLAS and makefile to build the library
<mkl directory>/interfaces/lapack95	Contains Fortran 95 wrappers for LAPACK and makefile to build the library

Table 3-1 High-level directory structure (continued)

Directory	Comment
<mkl directory>/interfaces/fftw2xc	Contains wrappers for FFTW version 2.x (C interface) to call Intel MKL FFTs
<mkl directory>/interfaces/fftw2xf	Contains wrappers for FFTW version 2.x (Fortran interface) to call Intel MKL FFTs
<mkl directory>/interfaces/fftw3xc	Contains wrappers for FFTW version 3.x (C interface) to call Intel MKL FFTs
<mkl directory>/interfaces/fftw3xf	Contains wrappers for FFTW version 3.x (Fortran interface) to call Intel MKL FFTs
<mkl directory> /interfaces/fftw2x_cdft	Contains wrappers for MPI FFTW version 2.x to call Intel MKL Cluster FFT interface
<mkl directory>/tests	Source and data for tests
<mkl directory>/lib/32	Contains static libraries and shared objects for IA-32 architecture
<mkl directory>/lib/em64t	Contains static libraries and shared objects for Intel® 64 architecture (formerly, Intel® EM64T)
<mkl directory>/lib/64	Contains static libraries and shared objects for IA-64 architecture (Itanium® processor family)
<mkl directory>/benchmarks/linpack	Contains OMP version of LINPACK benchmark
<mkl directory>/benchmarks/mp_linpack	Contains MPI version of LINPACK benchmark
<mkl directory>/tools/builder	Contains tools for creating custom dynamically linkable libraries
<mkl directory>/tools/environment	Contains shell scripts to set environmental variables in the user shell
<mkl directory>/tools/support	Contains a utility for reporting the package ID and license key information to Intel® Premier Support
<mkl directory>/tools/plugins/ com.intel.mkl.help	Contains an Eclipse plug-in with Intel MKL Reference Manual in WebHelp format. See Doc_Index.htm for comments.

Layered Model Concept

The Intel Math Kernel Library has long had a structure that is not visible to the user except for the 32-bit Windows* version. In that case, two interface libraries are provided, one of which the user needs to select at run time. Both libraries are relatively small and independent of the specific IA-32 architecture based processor. The use of these files makes it possible to support two different compiler interface standards without greatly increasing the size of the library, as duplication of most of the library, which is independent of the interface, is avoided.

Starting with release 10.0, Intel MKL is extending this approach to support a richer set of circumstances, compilers and threading in particular.

Interfaces. On Linux systems based on IA-64 architecture, the Intel® Fortran compiler returns complex values differently than gnu and certain other compilers do. Rather than duplicate the library for these differences, separate interface libraries are provided, which ease the support of differences between compilers while constraining the size of the library. Similarly, LP64 can be supported on top of ILP64 through an interface. Moreover, certain software vendors have requested support of legacy supercomputers where single precision means 64-bit arithmetic. Again, an interface library provides the needed mapping.

Threading. Intel MKL has long employed function-level threading throughout the library, choosing to avoid loop-level threading for efficiency reasons. Consequently, all the threading can be constrained to a relatively small set of functions and collected into a library. All references to compiler-specific run-time libraries are generated in these functions. Compiling them with different compilers and providing a threading library layer for each supported compiler permits Intel MKL to work in programs threaded with supported threading compilers other than compilers from Intel. As all threading is provided through OpenMP, but compiling this layer with threading is turned off, a non-threaded version of the library can also be provided through a layer without threading.

Computation. For any given processor family (processors based on IA-32, IA-64, or Intel® 64 architecture), a single computational library is used for all interfaces and threading layers, as there is no parallelism in the computational layer.

Run-time library (RTL). The last layer provides RTL support. Not all RTLs are delivered with Intel MKL. The only RTLs provided, except those that are relevant to the Intel MKL cluster software, are Intel® Compiler-based RTLs: Intel® Legacy OpenMP* run-time compiler library (`libguide`) and Intel® Compatibility OpenMP* run-time compiler library (`libiomp`). To thread using threading compilers other than those from Intel, you can employ Threading layer libraries or use the Compatibility library in the appropriate circumstances.

Layers

There are four essential parts of the library:

1. Interface layer.
2. Threading layer.
3. Computational layer.
4. Compiler Support RTL layer (RTL layer, for brevity).

Interface Layer. The layer essentially provides matching between the compiled code of your application and the threading and/or computational parts of the library. This layer may allow for matching like these:

- Provides LP64 interface to Intel MKL ILP64 software (see Support for ILP64 Programming for details)
- Provides means to deal with the way different compilers return function values.
- For those software vendors that use Cray-style names, provides mapping of single-precision names to double-precision ones in applications that employ ILP64.

Threading Layer. This layer provides a way for threaded Intel MKL to share supported compiler threading. The the layer also provides for a a sequential version of the library. What was internal to the library previously, now is essentially exposed in the threading layer by being compiled for different environments (threaded or sequential) and compilers (Intel, gnu, and so on).

Computational Layer. It is the heart of Intel MKL and has only one variant for any processor/operating system family, such as 32-bit Intel® processors on a 32-bit operating system. The computational layer can accommodate multiple architectures through identification of the architecture or architectural feature and choose the appropriate binary code at execution. Intel MKL may be thought of as the large computational layer that is unaffected by different computational environments. Then, as it has no RTL requirements, RTLs refer not to the computational layer but to one of the layers above it, that is, the interface layer or the threading layer. The most likely case is matching the threading layer with the RTL layer.

Compiler Support RTL Layer. This layer has run-time library support functions. For example, `libguide` and `libiomp` are RTLs providing threading support for the OpenMP* threading in Intel MKL.

See also the "Linking with Threading Libraries" section in chapter 5.

Sequential Version of the Library

Starting with release 9.1, the Intel MKL package provides support for sequential, that is, non-threaded, version of the library. It requires no Compiler Support RTL layer, that is, no Legacy OpenMP* or Compatibility OpenMP* run-time libraries, and does not respond to the environment variable `OMP_NUM_THREADS` (see the Using Intel MKL Parallelism section in chapter 6 for details). This version of Intel MKL runs unthreaded code. However, it is thread-safe, which means that you can use it in a parallel region from your own OpenMP code. You should use sequential version only if you have a particular reason not to use Intel MKL threading. The sequential version (layer) may be helpful when using Intel MKL with programs threaded with non-Intel compilers or in other situations where you may, for various reasons, need a non-threaded version of the library. For more information, see section Avoiding Conflicts in the Execution Environment in chapter 6.

To obtain sequential version of Intel MKL, in the Threading layer, choose the `*sequential.*` library to link (see Directory Structure in Detail).

Note that the sequential library depends on the POSIX threads library (`pthread`), which is used to make Intel MKL software thread-safe and should be included in the link line (see More Linking Examples in chapter 5).

Support for ILP64 Programming

The terms "LP64" and "ILP64" are used for certain historical reasons and due to the programming models philosophy described here:
http://www.unix.org/version2/whatsnew/lp64_wp.html

Intel MKL ILP64 libraries do not completely follow the programming models philosophy. However, the general idea is the same: use 64-bit integer type for indexing huge arrays, that is, arrays with more than 2^{31}-1 elements.

The LP64 and ILP64 interfaces are supported in the Interface layer. Once the appropriate library in the Interface layer is selected (see Directory Structure in Detail), all libraries below the Interface layer are compiled using the chosen interface.

As the differences between the ILP64 and LP64 interfaces are out of scope of the *Intel MKL Reference Manual*, you are encouraged to browse the include files, examples, and tests for the ILP64 interface details. To do this, see the following directories, respectively:

`<mkl directory>/include`

`<mkl directory>/examples`

`<mkl directory>/tests`

This section shows

- How the ILP64 concept is implemented specifically for Intel MKL
- How to compile your code for the ILP64 interface
- How to code for the ILP64 interface
- How to browse the Intel MKL include files for the ILP64 interface

This section also explains limitations of the ILP64 support.

Concept

ILP64 interface is provided for the following two reasons:

- To support huge data arrays, that is, arrays with more than 2 billion elements
- To enable compiling your Fortran code with the `-i8` compiler option.

Intel® Fortran compiler supports the -i8 option for changing behavior of the INTEGER type. By default the standard INTEGER type is 4-byte. The -i8 option makes the compiler treat INTEGER constants, variables, function and subroutine parameters as 8-byte.

The ILP64 binary interface uses 8-byte integers for function parameters that define array sizes, indices, strides, etc. At the language level, that is, in the *.f90 and *.fi files located in the Intel MKL include directory, such parameters are declared as INTEGER.

To bind your Fortran code with the ILP64 interface, you must compile your code with -i8 compiler option. And vice-versa, if your code is compiled with -i8, you can bind it only with the ILP64 interface, as the LP64 binary interface requires the INTEGER type to be 4-byte.

Note that some Intel MKL functions and subroutines have scalar or array parameters of type INTEGER*4 or INTEGER(KIND=4), which are always 4-byte, regardless of whether the code is compiled with the -i8 option.

For the languages of C and C++, Intel MKL provides the MKL_INT type as a counterpart of the INTEGER type for Fortran. MKL_INT is a macro defined as the standard C/C++ type int by default. However, if the MKL_ILP64 macro is defined for the code compilation, MKL_INT is defined as a 64-bit integer type. To define the MKL_ILP64 macro, you may call the compiler with the -DMKL_ILP64 command-line option.

Intel MKL also defines the type MKL_LONG for maintaining ILP64 interface in the specific case of FFT interface for C/C++. The MKL_LONG macro is defined as the standard C/C++ type long by default; and if the MKL_ILP64 macro is defined for the code compilation, MKL_LONG is defined as a 64-bit integer type.

NOTE. The type int is 32-bit for the Intel® C++ compiler, as well as for most of modern C/C++ compilers. The type long is 32- or 64-bit for the Intel® C++ and compatible compilers, depending on the particular OS.

In the Intel MKL interface for the C or C++ languages, that is, in the *.h header files located in the Intel MKL include directory, such function parameters as array sizes, indices, strides, etc. are declared as MKL_INT.

The FFT interface for C/C++ is the specific case. The header file mkl_dfti.h uses the MKL_LONG type for both explicit and implicit parameters of the interface functions. Specifically, type of the explicit parameter *dimension* of the function DftiCreateDescriptor() is MKL_LONG and type of the implicit parameter *length* is MKL_LONG for a one-dimensional transform and MKL_LONG[] (that is, an array of numbers having type MKL_LONG) for a multi-dimensional transform.

To bind your C or C++ code with the ILP64 interface, you must provide the `-DMKL_ILP64` command-line option to the compiler to enforce `MKL_INT` and `MKL_LONG` being 64-bit. And vice-versa, if your code is compiled with `-DMKL_ILP64` option, you can bind it only with the ILP64 interface, as the LP64 binary interface requires `MKL_INT` to be 32-bit and `MKL_LONG` to be the standard `long` type.

Note that certain MKL functions have parameters explicitly declared as `int` or `int[]`. Such integers are always 32-bit regardless of whether the code is compiled with the `-DMKL_ILP64` option.

Table 3-2 summarizes how the Intel MKL ILP64 concept is implemented:

Table 3-2 Intel MKL ILP64 concept

	Fortran	C or C++
The same include directory for ILP64 and LP64 interfaces	<*mkl directory*>/include	
Type used for parameters that are always 32-bit	`INTEGER*4`	`int`
Type used for parameters that are 64-bit integers for the ILP64 interface and 32-bit integers for LP64 interface	`INTEGER`	`MKL_INT`
Type used for all integer parameters of the FFT functions	`INTEGER`	`MKL_LONG`
Command-line option to control compiling for ILP64	`-i8`	`-DMKL_ILP64`

Compiling for ILP64

The same copy of the Intel MKL include directory is used for both ILP64 and LP64 interfaces. So, the compilation for the ILP64 interface looks like this:

Fortran:

```
ifort -i8 -I<mkl drectory>/include …
```

C or C++:

```
icc -DMKL_ILP64 -I<mkl directory>/include …
```

To compile for the LP64 interface, just omit the `-i8` or `-DMKL_ILP64` option.
Notice that linking of the application compiled with the `-i8` or `-DMKL_ILP64` option to the LP64 libraries may result in unpredictable consequences and erroneous output.

Table 3-3 summarizes of the compiler options:

Table 3-3 Compiler options for ILP64 interface

	Fortran	C or C++
Compiling for ILP64 interface	`ifort -i8 ...`	`icc -DMKL_ILP64 ...`
Compiling for LP64 interface	`ifort ...`	`icc ...`

Coding for ILP64

Although the `*.f90`, `*.fi`, and `*.h` files in the Intel MKL include directory were changed to meet requirements of the ILP64 interface, the LP64 interface was not changed. That is, all function parameters that were 32-bit integers still remain to have the 32-bit integer type, and all function parameters that were standard long integers still remain belonging to the standard `long` type. So, you do not need to change a single line of the existing code if you are not using the ILP64 interface.

To migrate to ILP64 or write new code for ILP64, you need to use appropriate types for parameters of the Intel MKL functions and subroutines. For the parameters that must be 64-bit integers in ILP64, you are encouraged to use the universal integer types, namely,

- `INTEGER` for Fortran
- `MKL_INT` for C/C++
- `MKL_LONG` for the parameters of the C/C++ FFT interface.

This way you make your code universal for both ILP64 and LP64 interfaces.

You may alternatively prefer to use other 64-bit types for the integer parameters that must be 64-bit in ILP64. For example, with Intel® compilers, you may use types:

- `INTEGER(KIND=8)` for Fortran
- `long long int` for C or C++

Note however that your code written this way will not work for the LP64 interface. Table 3-4 summarizes usage of the integer types.

Table 3-4 Integer types

	Fortran	C or C++

32-bit integers	`INTEGER*4`
	or
	`INTEGER(KIND=4)`
		`int`

Table 3-4 Integer types (continued)

	Fortran	C or C++
Universal integers: • 64-bit for ILP64 • 32-bit otherwise	INTEGER without specifying KIND	MKL_INT
Universal type for the FFT interface parameters	INTEGER without specifying KIND	MKL_LONG

Browsing Intel MKL include files

Gven a function with integer parameters, the *Reference Manual* does not explain which parameters become 64-bit and which remain 32-bit for ILP64.

To find out this information, you need to browse the include files, examples or tests. You are encouraged to start with browsing the include files, as they contain prototypes for all Intel MKL functions. Then you may see the examples and tests for better understanding of the function usage.

All include files are located in the *<mkl directory>*/include directory. Table 3-5 shows the include files to browse:

Table 3-5 Intel MKL include files

Function domain	Include files	
	Fortran	C or C++
BLAS Routines	mkl_blas.f90 mkl_blas.fi	mkl_blas.h
CBLAS Interface to BLAS		mkl_cblas.h
Sparse BLAS Routines	mkl_spblas.fi	mkl_spblas.h
LAPACK Routines	mkl_lapack.f90 mkl_lapack.fi	mkl_lapack.h
ScaLAPACK Routines		mkl_scalapack.h
Sparse Solver Routines		
• PARDISO	mkl_pardiso.f77 mkl_pardiso.f90	mkl_pardiso.h
• DSS Interface	mkl_dss.f77 mkl_dss.f90	mkl_dss.h
• RCI Iterative Solvers • ILU Factorization	mkl_rci.fi	mkl_rci.h
Optimization Solver Routines	mkl_rci.fi	mkl_rci.h

Table 3-5 Intel MKL include files (continued)

Function domain	Include files	
	Fortran	**C or C++**
Vector Mathematical Functions	`mkl_vml.fi`	`mkl_vml_functions.h`
Vector Statistical Functions	`mkl_vsl.fi` `mkl_vsl_subroutine.fi`	`mkl_vsl_functions.h`
Fourier Transform Functions	`mkl_dfti.f90`	`mkl_dfti.h`
Cluster Fourier Transform Functions	`mkl_cdft.f90`	`mkl_cdfti.h`
Partial Differential Equations Support Routines		
• Trigonometric Transforms	`mkl_trig_transforms.f90`	`mkl_trig_transforms.h`
• Poisson Solvers	`mkl_poisson.f90`	`mkl_poisson.h`

Some function domains that support only Fortran interface according to Table A-1, anyway provide header files for C or C++ in the include directory. Such `*.h` files enable using Fortran binary interface from C or C++ code and so describe the C interface, including its ILP64 aspect.

Limitations

Note that, not all components support the ILP64 feature. Table 3-6 shows which function domains support ILP64 interface.

Table 3-6 ILP64 support in Intel MKL

Function domain	Support for ILP64
BLAS	Yes
Sparse BLAS	Yes
LAPACK	Yes
ScaLAPACK	Yes
VML	Yes
VSL	Yes
PARDISO solvers	Yes
DSS solvers	Yes
ISS solvers	Yes
Optimization (Trust-Region) solvers	Yes
FFT	Yes

Table 3-6 ILP64 support in Intel MKL (continued)

Function domain	Support for ILP64
FFTW	No
Cluster FFT	Yes
PDE support: Trigonometric Transforms	Yes
PDE support: Poisson Solvers	Yes
GMP	No
Interval Arithmetic	No
BLAS 95	Yes
LAPACK 95	Yes

Intel MKL Versions

Intel MKL for Linux* distinguishes the following versions:

- for IA-32 architecture; the version is located in the `lib/32` directory.
- for Intel® 64 architecture; the version is located in the `lib/em64t` directory.
- for IA-64 architecture; the version is located in the `lib/64` directory.

See detailed structure of these directories in Table 3-7.

Directory Structure in Detail

The information in the table below shows detailed structure of the architecture-specific directories of the library. For the contents of the `doc` directory, see the Contents of the Documentation Directory subsection. See chapter 10 for the contents of subdirectories of the `benchmarks` directory.

Table 3-7 Detailed directory structure

Directory/file	Contents
`lib/32`[1]	Contains all libraries for IA-32 architecture
Static Libraries	
Interface layer	
`libmkl_intel.a`	Interface library for Intel® compiler
`libmkl_gf.a`	Interface library for GNU Fortran compiler

Table 3-7 Detailed directory structure (continued)

Directory/file	Contents
Threading layer	
libmkl_intel_thread.a	Parallel drivers library supporting Intel compiler
libmkl_gnu_thread.a	Parallel drivers library supporting GNU compiler
libmkl_sequential.a	Sequential drivers library
Computational layer	
libmkl_core.a	Kernel library for IA-32 architecture
libmkl_ia32.a	Dummy library. Contains references to Intel MKL libraries
libmkl_lapack.a	Dummy library. Contains references to Intel MKL libraries
libmkl_solver.a	Sparse Solver, Interval Solver, and GMP routines
libmkl_solver_sequential.a	Sequential version of Sparse Solver, Interval Solver, and GMP routines library
libmkl_scalapack.a	Dummy library. Contains references to Intel MKL libraries
libmkl_scalapack_core.a	ScaLAPACK routines
libmkl_cdft.a	Dummy library. Contains references to Intel MKL libraries
libmkl_cdft_core.a	Cluster version of FFTs
RTL layer	
libguide.a	Intel® Legacy OpenMP* run-time library for static linking
libiomp5.a	Intel® Compatibility OpenMP* run-time library for static linking
libmkl_blacs.a	BLACS routines supporting the following MPICH versions: • Topspin* MPICH version 1.2.5 configured with *ch_vapi* device • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2
libmkl_blacs_intelmpi.a	BLACS routines supporting Intel MPI 1.0
libmkl_blacs_intelmpi20.a	BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH 2.0
libmkl_blacs_openmpi.a	BLACS routines supporting OpenMPI.
Dynamic Libraries	
Interface layer	
libmkl_intel.so	Interface library for Intel® compiler

Table 3-7 Detailed directory structure (continued)

Directory/file	Contents
libmkl_gf.so	Interface library for GNU Fortran compiler
Threading layer	
libmkl_intel_thread.so	Parallel drivers library supporting Intel compiler
libmkl_gnu_thread.so	Parallel drivers library supporting GNU compiler
libmkl_sequential.so	Sequential drivers library
Computational layer	
libmkl.so	Dummy library. Contains references to Intel MKL libraries
libmkl_core.so	Library dispatcher for dynamic load of processor-specific kernel library
libmkl_def.so	Default kernel library (Intel® Pentium®, Pentium® Pro, and Pentium® II processors)
libmkl_p3.so	Intel® Pentium® III processor kernel library
libmkl_p4.so	Pentium® 4 processor kernel library
libmkl_p4p.so	Kernel library for Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3)
libmkl_p4m.so	Kernel library for processors based on the Intel® Core™ microarchitecture (except Intel® Core™ Duo and Intel® Core™ Solo processors, for which mkl_p4p.so is intended)
libmkl_lapack.so	LAPACK routines and drivers
libmkl_ias.so	Interval arithmetic routines
libmkl_vml_def.so	VML/VSL part of default kernel for old Intel® Pentium® processors
libmkl_vml_ia.so	VML/VSL default kernel for newer Intel® architecure processors
libmkl_vml_p3.so	VML/VSL part of Pentium® III processor kernel
libmkl_vml_p4.so	VML/VSL part of Pentium® 4 processor kernel
libmkl_vml_p4p.so	VML/VSL for Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3)
libmkl_vml_p4m.so	VML/VSL for processors based on the Intel® Core™ microarchitecture
libmkl_vml_p4m2.so	VML/VSL for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
RTL layer	
libguide.so	Intel® Legacy OpenMP* run-time library for dynamic linking

Table 3-7 Detailed directory structure (continued)

Directory/file	Contents
libiomp5.so	Intel® Compatibility OpenMP* run-time library for dynamic linking
lib/em64t[1]	Contains all libraries for Intel® 64 architecture
Static Libraries	
Interface layer	
libmkl_intel_ilp64.a	ILP64 interface library for Intel compiler
libmkl_intel_lp64.a	LP64 interface library for Intel compiler
libmkl_intel_sp2dp.a	SP2DP interface library for Intel compiler
libmkl_gf_ilp64.a	ILP64 interface library for GNU Fortran compiler
libmkl_gf_lp64.a	LP64 interface library for GNU Fortran compiler
Threading layer	
libmkl_intel_thread.a	Parallel drivers library supporting Intel compiler
libmkl_gnu_thread.a	Parallel drivers library supporting GNU compiler
libmkl_sequential.a	Sequential drivers library
Computational layer	
libmkl_core.a	Kernel library for Intel® 64 architecture
libmkl_em64t.a	Dummy library. Contains references to Intel MKL libraries
libmkl_lapack.a	Dummy library. Contains references to Intel MKL libraries
libmkl_solver.a	Dummy library. Contains references to Intel MKL libraries
libmkl_solver_lp64.a	Sparse Solver, Interval Solver, and GMP routines library supporting LP64 interface
libmkl_solver_ ilp64.a	Sparse Solver routines library supporting ILP64 interface
libmkl_solver_lp64_ sequential.a	Sequential version of Sparse Solver, Interval Solver, and GMP routines library supporting LP64 interface
libmkl_solver_ilp64_ sequential.a	Sequential version of Sparse Solver routines library supporting ILP64 interface
libmkl_scalapack.a	Dummy library. Contains references to Intel MKL libraries
libmkl_scalapack_ lp64.a	ScaLAPACK routines library supporting LP64 interface
libmkl_scalapack_ ilp64.a	ScaLAPACK routines library supporting ILP64 interface
libmkl_cdft.a	Dummy library. Contains references to Intel MKL libraries

Table 3-7 Detailed directory structure (continued)

Directory/file	Contents
libmkl_cdft_core.a	Cluster version of FFTs
RTL layer	
libguide.a	Intel® Legacy OpenMP* run-time library for static linking
libiomp5.a	Intel® Compatibility OpenMP* run-time library for static linking
libmkl_blacs_ilp64.a	ILP64 version of BLACS routines supporting the following MPICH versions: • Topspin* MPICH version 1.2.5 configured with *ch_vapi* device • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2
libmkl_blacs_lp64.a	LP64 version of BLACS routines supporting the following MPICH versions: • Topspin* MPICH version 1.2.5 configured with *ch_vapi* device • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2
libmkl_blacs_intelmpi_ilp64.a	ILP64 version of BLACS routines supporting Intel MPI 1.0
libmkl_blacs_intelmpi_lp64.a	LP64 version of BLACS routines supporting Intel MPI 1.0
libmkl_blacs_intelmpi20_ilp64.a	ILP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH 2.0
libmkl_blacs_intelmpi20_lp64.a	LP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH 2.0
libmkl_blacs_openmpi_ilp64.a	ILP64 version of BLACS routines supporting OpenMPI.
libmkl_blacs_openmpi_lp64.a	LP64 version of BLACS routines supporting OpenMPI.
Dynamic Libraries	
Interface layer	
libmkl_intel_ilp64.so	ILP64 interface library for Intel compiler
libmkl_intel_lp64.so	LP64 interface library for Intel compiler
libmkl_intel_sp2dp.so	SP2DP interface library for Intel compiler
libmkl_gf_ilp64.so	ILP64 interface library for GNU Fortran compiler
libmkl_gf_lp64.so	LP64 interface library for GNU Fortran compiler

Table 3-7 Detailed directory structure (continued)

Directory/file	Contents
Threading layer	
libmkl_intel_thread.so	Parallel drivers library supporting Intel compiler
libmkl_gnu_thread.so	Parallel drivers library supporting GNU compiler
libmkl_sequential.so	Sequential drivers library
Computational layer	
libmkl.so	Dummy library. Contains references to Intel MKL libraries
libmkl_core.so	Library dispatcher for dynamic load of processor-specific kernel
libmkl_def.so	Default kernel library
libmkl_p4n.so	Kernel library for Intel® Xeon® processor using Intel® 64 architecture
libmkl_mc.so	Kernel library for processors based on the Intel® Core™ microarchitecture
libmkl_lapack.so	LAPACK routines and drivers
libmkl_ias.so	Interval arithmetic routines
libmkl_vml_def.so	VML/VSL part of default kernels
libmkl_vml_mc.so	VML/VSL for processors based on the Intel® Core™ microarchitecture
libmkl_vml_p4n.so	VML/VSL for Intel® Xeon® processor using Intel® 64 architecture
libmkl_vml_mc2.so	VML/VSL for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
RTL layer	
libguide.so	Intel® Legacy OpenMP* run-time library for dynamic linking
libiomp5.so	Intel® Compatibility OpenMP* run-time library for dynamic linking
lib/64[1]	Contains all libraries for IA-64 architecture
Static Libraries	
Interface layer	
libmkl_intel_ilp64.a	ILP64 interface library for Intel compiler
libmkl_intel_lp64.a	LP64 interface library for Intel compiler
libmkl_intel_sp2dp.a	SP2DP interface library for Intel compiler
libmkl_gf_ilp64.a	ILP64 interface library for GNU Fortran compiler
libmkl_gf_lp64.a	LP64 interface library for GNU Fortran compiler

Table 3-7 Detailed directory structure (continued)

Directory/file	Contents
Threading layer	
libmkl_intel_thread.a	Parallel drivers library supporting Intel compiler
libmkl_gnu_thread.a	Parallel drivers library supporting GNU compiler
libmkl_sequential.a	Sequential drivers library
Computational layer	
libmkl_core.a	Kernel library for IA-64 architecture
libmkl_ipf.a	Dummy library. Contains references to Intel MKL libraries
libmkl_lapack.a	Dummy library. Contains references to Intel MKL libraries
libmkl_solver.a	Dummy library. Contains references to Intel MKL libraries
libmkl_solver_lp64.a	Sparse Solver, Interval Solver, and GMP routines library supporting LP64 interface
libmkl_solver_ ilp64.a	Sparse Solver routines library supporting ILP64 interface
libmkl_solver_lp64_ sequential.a	Sequential version of Sparse Solver, Interval Solver, and GMP routines library supporting LP64 interface
libmkl_solver_ilp64_ sequential.a	Sequential version of Sparse Solver routines library supporting ILP64 interface
libmkl_scalapack.a	Dummy library. Contains references to Intel MKL libraries
libmkl_scalapack_ lp64.a	ScaLAPACK routines library supporting LP64 interface
libmkl_scalapack_ ilp64.a	ScaLAPACK routines library supporting ILP64 interface
libmkl_cdft.a	Dummy library. Contains references to Intel MKL libraries
libmkl_cdft_core.a	Cluster version of FFTs
RTL layer	
libguide.a	Intel® Legacy OpenMP* run-time library for static linking
libiomp5.a	Intel® Compatibility OpenMP* run-time library for static linking
libmkl_blacs_ilp64.a	ILP64 version of BLACS routines supporting the following MPICH versions: • Topspin* MPICH version 1.2.5 configured with *ch_vapi* device • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2

Table 3-7 Detailed directory structure (continued)

Directory/file	Contents
libmkl_blacs_lp64.a	LP64 version of BLACS routines supporting the following MPICH versions: • Topspin* MPICH version 1.2.5 configured with *ch_vapi* device • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2
libmkl_blacs_intelmpi_ilp64.a	ILP64 version of BLACS routines supporting Intel MPI 1.0
libmkl_blacs_intelmpi_lp64.a	LP64 version of BLACS routines supporting Intel MPI 1.0
libmkl_blacs_intelmpi20_ilp64.a	ILP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH 2.0
libmkl_blacs_intelmpi20_lp64.a	LP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH 2.0
libmkl_blacs_openmpi_ilp64.a	ILP64 version of BLACS routines supporting OpenMPI.
libmkl_blacs_openmpi_lp64.a	LP64 version of BLACS routines supporting OpenMPI.
Dynamic Libraries	
Interface layer	
libmkl_intel_ilp64.so	ILP64 interface library for Intel compiler
libmkl_intel_lp64.so	LP64 interface library for Intel compiler
libmkl_intel_sp2dp.so	SP2DP interface library for Intel compiler
libmkl_gf_ilp64.so	ILP64 interface library for GNU Fortran compiler
libmkl_gf_lp64.so	LP64 interface library for GNU Fortran compiler
Threading layer	
libmkl_intel_thread.so	Parallel drivers library supporting Intel compiler
libmkl_gnu_thread.so	Parallel drivers library supporting GNU compiler
libmkl_sequential.so	Sequential drivers library
Computational layer	
libmkl.so	Dummy library. Contains references to Intel MKL libraries
libmkl_core.so	Library dispatcher for dynamic load of processor-specific kernel library
libmkl_i2p.so	Kernel library for IA-64 architecture

Table 3-7 Detailed directory structure (continued)

Directory/file	Contents
libmkl_lapack.so	LAPACK routines and drivers
libmkl_ias.so	Interval arithmetic routines
libmkl_vml_i2p.so	VML kernel for IA-64 architecture
RTL layer	
libguide.so	Intel® Legacy OpenMP* run-time library for dynamic linking
libiomp5.so	Intel® Compatibility OpenMP* run-time library for dynamic linking

1. Additionally, a number of interface libraries may be generated as a result of respective makefile operation in the `interfaces` directory (see "Using Language-Specific Interfaces with Intel MKL" section in chapter 7).

Dummy Libraries

Pure layered libraries give more flexibility to choose the appropriate combination of libraries but do not have backward compatibility by library names in link lines. Dummy libraries are introduced to provide backward compatibility with earlier version of MKL, which did not use layered libraries.

Dummy libraries do not contain any functionality, but only dependencies on a set of layered libraries. Placed in link line, dummy libraries enable omitting dependent layered libraries, which will be linked automatically. Dummy libraries contain dependency on the following layered libraries (default principle):

- Interface: Intel, LP64
- Threading: Intel compiled
- Computational: the computation library.

So, if you employ the above interface and use OpenMP* threading provided by the Intel® compiler, you may not change your link lines.

Contents of the Documentation Directory

Table 3-8 shows the contents of the `doc` subdirectory in the Intel MKL installation directory:

Table 3-8 Contents of the doc directory

File name	Comment
mklEULA.txt	Intel MKL license
mklSupport.txt	Information on package number for customer support reference

Table 3-8 Contents of the doc directory (continued)

File name	Comment
Doc_index.htm	Index of Intel MKL documentation
fftw2xmkl_notes.htm	FFTW 2.x Interface Support Technical User Notes
fftw3xmkl_notes.htm	FFTW 3.x Interface Support Technical User Notes
Install.txt	Installation Guide
mklman.pdf	Intel MKL Reference Manual
mklman90_j.pdf	Intel MKL Reference Manual in Japanese
Readme.txt	Initial User Information
redist.txt	List of redistributable files
Release_Notes.htm	Intel MKL Release Notes (HTML format)
Release_Notes.txt	Intel MKL Release Notes (text format)
vmlnotes.htm	General discussion of VML
vslnotes.pdf	General discussion of VSL
userguide.pdf	Intel MKL for Linux* User's Guide, this document.
./tables	Directory that contains tables referenced in vmlnotes.htm.

Configuring Your Development Environment

4

This chapter explains how to configure your development environment for the use with Intel® Math Kernel Library (Intel® MKL) and especially what features may be customized using the Intel MKL configuration file.

For information on how to set up environment variables for threading, refer to Setting the Number of Threads Using OpenMP Environment Variable section in Chapter 6.

Setting Environment Variables

After the installation of Intel MKL for Linux* is complete, you can use three scripts `mklvars32`, `mklvarsem64t`, and `mklvars64` with two flavors each (`.sh` and `.csh`) in the `tools/environment` directory to set the environment variables `INCLUDE`, `LD_LIBRARY_PATH`, `MANPATH`, `CPATH`, `FPATH`, and `LIBRARY_PATH` in the user shell. Section Automating the Process explains how to automate setting of these variables at startup.

If you want to further customize some of the Intel MKL features, you may use the configuration file `mkl.cfg`, which contains several variables that can be changed.

Automating the Process

To automate setting of these environment variables at startup, execution of the `mklvars*.sh` can be added to your shell profile so that each time you log in, the path to the appropriate Intel MKL directories will be set.

With the local user account, you should edit the following files by adding execution of the appropriate script to section "Path manipulation" right before exporting variables. The commands to be added should be like this:

- bash:

  `~/.bash_profile, ~/.bash_login or ~/.profile`

```
# setting up MKL environment for bash
. <absolute_path_to_installed_MKL>/tools/environment/mklvars<arch>.sh
```

- sh:

```
~/.profile

# setting up MKL environment for sh
. <absolute_path_to_installed_MKL>/tools/environment/mklvars<arch>.sh
```

- csh:

```
~/.login

# setting up MKL environment for csh
. <absolute_path_to_installed_MKL>/tools/environment/mklvars<arch>.csh
```

In the above commands, `mklvars<arch>` stands for each of `mklvars32`, `mklvarsem64t` or `mklvars64`.

If you have super user permissions, you can add the same commands to a general-system file in `/etc/profile` (for bash and sh) or in `/etc/csh.login` (for csh).

Before uninstalling Intel MKL, remove the above commands from all profile files where the script execution was added, to avoid problems during logging in.

Configuring Eclipse CDT to Link with Intel MKL

This section describes how to configure Eclipse C/C++ Development Tools (CDT) 3.x and 4.0 to link with Intel MKL.

TIP. After linking your CDT with Intel MKL, you can benefit from the Eclipse-provided *code assist* feature. See *Code/Context Assist* description in *Eclipse Help*.

Configuring Eclipse CDT 4.0

To configure Eclipse CDT 4.0 to link with Intel MKL, follow the instructions below:

1. If the tool-chain/compiler integration supports include path options, go to the **Includes** tab of the **C/C++ General > Paths and Symbols** property page and set the Intel MKL include path, for example, the default value is `/opt/intel/mkl/10.0.xxx/include`, where `xxx` is the Intel MKL package number, such as `"039"`.
2. If the tool-chain/compiler integration supports library path options, go to the **Library Paths** tab of the **C/C++ General > Paths and Symbols** property page and set a path to the Intel MKL libraries, depending upon the target architecture, for example, with the default installation, `/opt/intel/mkl/10.0.xxx/lib/em64t`
3. For a particular build, go to the **Tool Settings** tab of the **C/C++ Build > Settings** property page and specify names of the Intel MKL libraries to link with your application, for example, `mkl_solver_lp64` and `mkl_core` (As compilers typically require library names rather than library file names, `the` "`lib`" prefix and "`a`" extension are omitted). See section "Selecting Libraries to Link" in chapter 5 on the choice of the libraries. The name of the particular setting where libraries are specified depends upon the compiler integration.

Note that the compiler/linker will automatically pick up the include and library paths settings only in case the automatic makefile generation is turned on, otherwise, you will have to specify the include and library paths directly in the makefile to be used.

Configuring Eclipse CDT 3.x

To configure Eclipse CDT 3.x to link with Intel MKL, follow the instructions below:

- For Standard Make projects,
  1. Go to **C/C++ Include Paths and Symbols** property page and set the Intel MKL include path, for example, the default value is `/opt/intel/mkl/10.0.xxx/include` where `xxx` is the Intel MKL package number, for instance, `"039"`.
  2. Go to the **Libraries** tab of the **C/C++ Project Paths** property page and set the Intel MKL libraries to link with your applications, for example, `/opt/intel/mkl/10.0.xxx/lib/em64t/libmkl_lapack.a` and `/opt/intel/mkl/10.0.xxx/lib/em64t/libmkl_em64t.a`. See section "Selecting Libraries to Link" in chapter 5 on the choice of the libraries.

  Note that with the Standard Make, the above settings are needed for the CDT internal functionality only. The compiler/linker will not automatically pick up these settings and you will still have to specify them directly in the makefile.
- For Managed Make projects, you can specify settings for a particular build. To do this,

1. Go to the **Tool Settings** tab of the **C/C++ Build** property page. All the settings you need to specify are on this page. Names of the particular settings depend upon the compiler integration and therefore are not given below.
2. If the compiler integration supports include path options, set the Intel MKL include path, for example, the default value is `/opt/intel/mkl/10.0.xxx/include`.
3. If the compiler integration supports library path options, set a path to the Intel MKL libraries, depending upon the target architecture, for example, with the default installation, `/opt/intel/mkl/10.0.xxx/lib/em64t`.
4. Specify names of the Intel MKL libraries to link with your application, for example, `mkl_lapack` and `mkl_ia32` (As compilers typically require library names rather than library file names, `the "lib"` prefix and `"a"` extension are omitted). See section "Selecting Libraries to Link" in chapter 5 on the choice of the libraries.

1. Make sure that your project that uses Intel MKL is open and active.

Customizing the Library Using the Configuration File

Intel MKL configuration file provides the possibility to redefine names of dynamic libraries.

You may create a configuration file with the `mkl.cfg` name to assign values to a number of variables. Below is an example of the configuration file containing all possible variables with their default values:

Example 4-1 Intel MKL configuration file

```
//
// Default values for mkl.cfg file
//
// SO names for IA-32 architecture
MKL_X87so = mkl_def.so
MKL_SSE1so = mkl_p3.so
MKL_SSE2so = mkl_p4.so
MKL_SSE3so = mkl_p4p.so
MKL_VML_X87so = mkl_vml_def.so
MKL_VML_SSE1so = mkl_vml_p3.so
MKL_VML_SSE2so = mkl_vml_p4.so
MKL_VML_SSE3so = mkl_vml_p4p.so
```

Example 4-1 Intel MKL configuration file (continued)

```
// SO names for Intel(R) 64 architecture
MKL_EM64TDEFso = mkl_def.so
MKL_EM64TSSE3so = mkl_p4n.so
MKL_VML_EM64TDEFso = mkl_vml_def.so
MKL_VML_EM64TSSE3so = mkl_vml_p4n.so
// SO names for Intel(R) Itanium(R) processor family
MKL_I2Pso = mkl_i2p.so
MKL_VML_I2Pso = mkl_vml_i2p.so
// SO name for LAPACK library
MKL_LAPACKso = mkl_lapack.so
```

When any Intel MKL function is first called, Intel MKL checks to see if the configuration file exists, and if so, it operates with the specified names. The path to the configuration file is specified by environment variable `MKL_CFG_FILE`. If this variable is not defined, then first the current directory is searched through, and then the directories specified in the `PATH` environment variable. If the Intel MKL configuration file does not exist, the library operates with standard names of libraries.

If the variable is not specified in the configuration file, or specified incorrectly, standard names of libraries are used.

Below is an example of the configuration file, which redefines the library names:

Example 4-2 Redefining library names using the configuration file

```
// SO redefinition
MKL_X87so = matlab_x87.so
MKL_SSE1so = matlab_sse1.so
MKL_SSE2so = matlab_sse2.so
MKL_SSE3so = matlab_sse2.so
MKL_ITPso = matlab_ipt.so
MKL_I2Pso = matlab_i2p.so
```

Linking Your Application with Intel® Math Kernel Library

5

This chapter features linking of your applications with Intel® Math Kernel Library (Intel® MKL) for Linux*. The chapter compares static and dynamic linking models; describes the general link line syntax to be used for linking with Intel MKL libraries; provides comprehensive information in a tabular form on the libraries that should be linked with your application for your particular platform and function domain; gives linking examples. Building of custom shared objects is also discussed.

Selecting Between Linkage Models

You can link your applications with Intel MKL libraries statically, using static library versions, or dynamically, using shared libraries.

Static Linking

With static linking, all links are resolved at link time. Therefore, the behavior of statically built executables is absolutely predictable, as they do not depend upon a particular version of the libraries available on the system where the executables run. Such executables must behave exactly the same way as was observed during testing. The main disadvantage of static linking is that upgrading statically linked applications to higher library versions is troublesome and time-consuming, as you have to relink the entire application. Besides, static linking produces large-size executables and uses memory inefficiently, since if several executables are linked with the same library, each of them loads it into memory independently. However, this is hardly an issue for Intel MKL, used mainly for large-size problems. — It matters only for executables having data size relatively small and comparable with the size of the executable.

Dynamic Linking

During dynamic linking, resolving of some undefined symbols is postponed until run time. Dynamically built executables still contain undefined symbols along with lists of libraries that provide definitions of the symbols. When the executable is loaded, final linking is done before the application starts running. If several dynamically built executables use the same library, the library loads to memory only once and the executables share it, thereby saving memory. Dynamic linking ensures consistency in using and upgrading libraries, as all the dynamically built applications share the same library. This way of linking enables you to separately update libraries and applications that use the libraries, which facilitates keeping applications up-to-date. The advantages of dynamic linking are achieved at the cost of run-time performance losses, as a part of linking is done at run time and every unresolved symbol has to be looked up in a dedicated table and resolved. However, this is hardly an issue for Intel MKL.

Making the Choice

It is up to you to select whether to link in Intel MKL libraries dynamically or statically when building your application.

In most cases, users choose dynamic linking due to its strong advantages.

However, if you are developing applications to be shipped to a third-party, to have nothing else than your application shipped, you have to use static linking. To reduce the size of executables shipped, you can also build custom dynamic libraries (see Building Custom Shared Objects).

Table 5-1 compares the linkage models.

Table 5-1 Quick comparison of Intel MKL linkage models

Feature	Dynamic Linkage	Static Linkage	Custom Dynamic Linkage
Processor Updates	Automatic	Automatic	Recompile and redistribute
Optimization	All processors	All processors	All processors
Build	Link to dynamic libraries	Link to static libraries	Build separate dynamic libraries and link to them.
Calling	Regular names	Regular names	Modified names
Total Binary Size	Large	Small	Small
Executable Size	Smallest	Small	Smallest

Table 5-1 Quick comparison of Intel MKL linkage models (continued)

Feature	Dynamic Linkage	Static Linkage	Custom Dynamic Linkage
Multi-threaded / thread safe	Yes	Yes	Yes

Intel MKL-specific Linking Recommendations

You are strongly encouraged to dynamically link in Intel® Legacy OpenMP* run-time library `libguide` and Intel® Compatibility OpenMP* run-time library `libiomp`. Linking to static OpenMP run-time library is not recommended, as it is very easy with layered software to link in more than one copy of the library. This causes performance problems (too many threads) and may cause correctness problems if more than one copy is initialized.

You are advised to link with `libguide` and `libiomp` dynamically even if other libraries are linked statically.

Link Command Syntax

To link libraries having filenames `libyyy.a` or `libyyy.so` with your application, two options are available:

- In the link line, list library filenames using relative or absolute paths, for example:

  ```
  <ld> myprog.o /opt/intel/mkl/10.0.xxx/lib/32/libmkl_solver.a
  /opt/intel/mkl/10.0.xxx/lib/32/libmkl_intel.a
  /opt/intel/mkl/10.0.xxx/lib/32/libmkl_intel_thread.a
  /opt/intel/mkl/10.0.xxx/lib/32/libmkl_core.a
  /opt/intel/mkl/10.0.xxx/lib/32/libguide.so -lpthread
  ```

  where *<ld>* is a linker, `myprog.o` is the user's object file, and `xxx` is the Intel MKL package number, for example, `"039"`.
  Appropriate Intel MKL libraries are listed first and followed by the system library `libpthread`.
- In the link line, list library names (with absolute or relative paths, if needed) preceded with `-L`*<path>*, which indicates where to search for binaries, and `-I`*<include>*, which indicates where to search for header files. Discussion of linking with Intel MKL libraries employs this option.

To link with the Intel MKL libraries, follow this general form of specifying paths and libraries in the link line:

```
-L<MKL path> -I<MKL include>
```

```
[-lmkl_lapack95] [-lmkl_blas95]
[cluster components]
[-Wl,--start-group]
[{-lmkl_{intel, intel_ilp64, intel_lp64, intel_sp2dp, gf, gf_ilp64, gf_lp64}]
[-lmkl_{intel_thread, sequential}]
[{-lmkl_solver, -lmkl_solver_lp64, -lmkl_solver_ilp64}]
{{[-lmkl_lapack] -lmkl_{ia32, em64t, ipf}},
-lmkl_core}}
[-Wl,--end-group]
[{-lguide, -liomp5}] [-lpthread] [-lm]
```

See Selecting Libraries to Link for details of this syntax usage and specific recommendations on which libraries to link depending on your Intel MKL usage scenario. See also

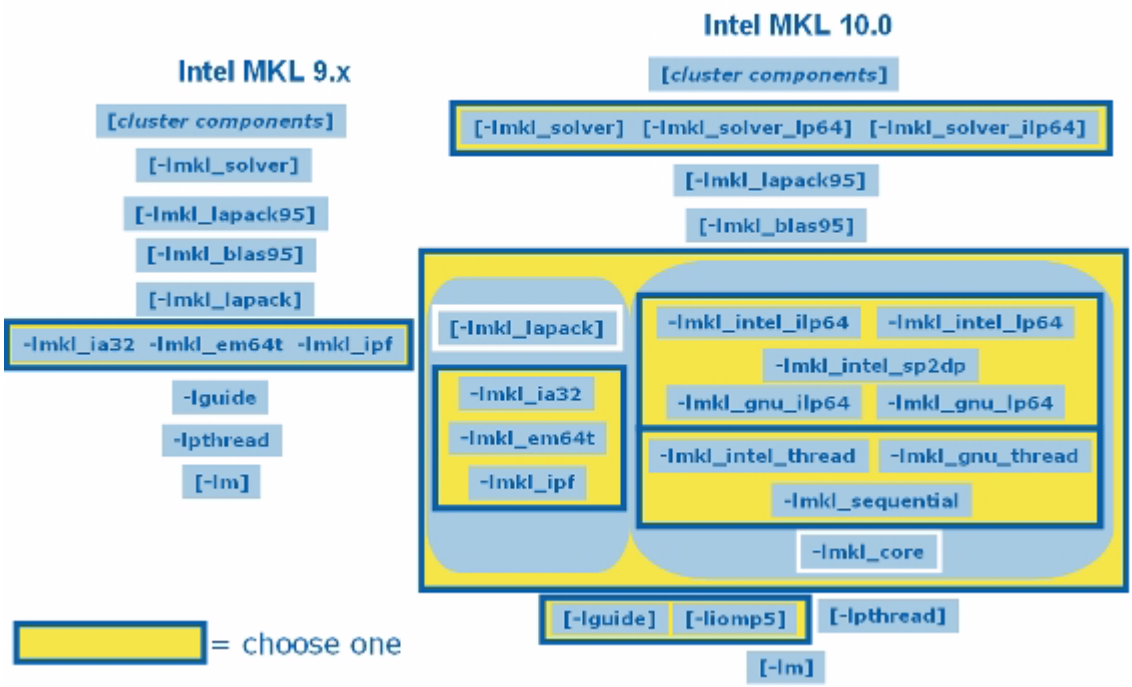
- section "Fortran 95 Interfaces and Wrappers to LAPACK and BLAS" in chapter 7 for information on the libraries that you should build prior to linking
- chapter Working with Intel® Math Kernel Library Cluster Software on lining with cluster components.

To link with Intel MKL, you can choose *pure layered mode*l or *default model*, which is backward compatible on link line (except cluster components). The syntax above incorportates both models.

For the pure layered model, you need to choose one library from the Interface layer, one library from the Threading layer, the Computational layer library (no choice here), and add run-time libraries. In case of the default model, you need not change the link line (see the Dummy Libraries section in cahpter 3 for details).

Figure 5-1 compares linking for Intel MKL 10.0 that uses layers and Intel MKL 9.x.

Figure 5-1 Linking with Layered Intel MKL



In case of employing the pure layered model for static linking, the interface layer, threading layer, and computation layer libraries must be enclosed in grouping symbols (for example, `-Wl,--start-group $MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group`).

In case you use dummy libraries,

- The path to Intel MKL libraries must be added to the list of paths that the linker will search for archive libraries (for example, as `-L<MKL path>`)
- No interface layer or threading layer libraries should be included in the link line
- No grouping symbols should be employed.

The order of listing libraries in the link line is essential, except for the libraries enclosed in the grouping symbols.

Selecting Libraries to Link

Below are several simple examples of link libraries for the layered pure and layered default link models on 64-bit Linux* based on Intel® 64 architecture for different components using Intel® compiler interface.

- BLAS, FFT, VML, VSL components, static case:

  Layered default: `libmkl_em64t.a`

  Layered pure: `libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`
- BLAS, FFT, VML, VSL components, dynamic case:

  Layered default: `libmkl.so`

  Layered pure: `libmkl_intel_lp64.so libmkl_intel_thread.so libmkl_core.so`
- LAPACK, static case:

  Layered default: `libmkl_lapack.a libmkl_em64t.a`

  Layered pure: `libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`
- LAPACK, dynamic case:

  Layered default:`libmkl_lapack.so libmkl.so`

  Layered pure: `libmkl_intel_lp64.so libmkl_intel_thread.so libmkl_core.so`
- ScaLAPACK, static case:

  Layered default: `libmkl_scalapack.a libmkl_blacs.a libmkl_lapack.a libmkl_em64t.a`

  Layered pure: `libmkl_intel_lp64.a libmkl_scalapack_core.a libmkl_blacs.a libmkl_intel_thread.a libmkl_core.a`
- PARDISO, static case:

  Layered default: `libmkl_solver.a libmkl_lapack.a libmkl_em64t.a`

  Layered pure, LP64: `libmkl_solver_lp64.a libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`

  Layered pure, ILP64: `libmkl_solver_ilp64.a libmkl_intel_ilp64.a libmkl_intel_thread.a libmkl_core.a`

When linking (see Link Command Syntax and More Linking Examples), note that

- The solver library currently does not comply with the layered model. So, it is not changed internally with respect to the Intel MKL 9.x. However, to support LP64/ILP64 interfaces, two libraries were introduced in the unified structure: `libmkl_solver_lp64.a` for the LP64 interface and `libmkl_solver_ilp64.a` for the ILP64 interface. For backward link line compatibility `libmkl_solver.a` has become a dummy library. There is still only static version of the solver library, as it was for previous releases. To link with the solver library using the pure layered model, you should also include the library `libmkl_solver_lp64.a` or `libmkl_solver_ilp64.a` in the link line, depending upon the interface you need.

- `libmkl_lapack95.a` and `libmkl_blas95.a` libraries contain LAPACK95 and BLAS95 interfaces respectively. They are not included into the original distribution and should be built before using the interface. (See "Fortran 95 Interfaces and Wrappers to LAPACK and BLAS" section in chapter 7 for details on building the libraries and "Compiler-dependent Functions and Fortran 95 Modules" section on why *source code* is distributed in this case.)
- If you use the Intel MKL FFT, VML, or VSL, you will need to link in the Linux math support library `libm` by adding "`-lm`" to the link line.
- In products for Linux, it is necessary to link the `pthreads` library by adding `-lpthread`. The `pthread` library is native to Linux and `libguide` makes use of this library to support multi-threading. Any time `libguide` is required (threaded software from Intel MKL is used) you must add `-lpthread` at the end of your link line (link order is important).

Linking with Threading Libraries

In the past, only few compilers other than Intel® ones supported threading of the user's application. Starting with Intel MKL 10.0 timeframe, additional compilers will be offering OpenMP* threading. If an application compiled with such a threading compiler used OpenMP threading and called threaded parts of Intel MKL versions lower than 10.0, there might be difficulties. They may arise because MKL is threaded using the Intel® compilers, and threading libraries from different compilers are not compatible. This can lead to performance issues, and perhaps even failures when incompatible threading is used within the same application. Starting with Intel MKL 10.0, several solutions are available in certain cases. Those solutions are provided both from the Threading Layer and the supplied run-time libraries found in the Compiler Support RTL Layer.

The Solution in Layers. With this release of Intel MKL, the library is structured as layers. One of those layers is a Threading Layer. Because of the internal structure of the library, all of the threading represents a small amount of code. This code is compiled by different compilers (such as the gnu compiler on Linux*) and the appropriate layer linked in with the threaded application.

The second relevant component is the Compiler Support RTL Layer. Prior to Intel MKL 10.0, this layer only included the Intel® Legacy OpenMP* run-time compiler library `libguide`. Now you have a new choice to use the Intel® Compatibility OpenMP* run-time compiler library `libiomp`. The Compatibility library provides support for one additional threading compiler on Linux (gnu). That is, a program threaded with gnu compiler can safely be linked with intel MKL and `libiomp` and execute efficiently and effectively.

More about libiomp. `libiomp` is a new software. It has successfully been through a beta trial, it is robust and has shown few bugs even in the beta. In addition it offers excellent scaling with increasing numbers of cores in comparison to the Microsoft or gnu threading libraries. `libiomp` is essentially `libguide` with an interface layer to map the compiler generated function calls to the `libguide` thread management software.

Table 5-2 shows different scenarios, depending on the threading compiler used, and the possibilities for each scenario to choose the Threading layer and RTL layer when using the current version of Intel MKL (static cases only):

Table 5-2 Selecting the Threading and RTL layer

Compiler	Application Threaded?	Threading Layer	RTL Layer Recommended	Comment
Intel	Does not matter	`mkl_intel_thread.`	`libguide.so` or `libiomp5.so`	
gnu	Yes	`libmkl_gnu_thread.a`	`libiomp5.so` or GNU OpenMP layer	`libiomp5` offers superior scaling performance
gnu	Yes	`libmkl_sequential.a`	None	
gnu	No	`libmkl_intel_thread.a`	`libguide.so` or `libiomp5.so`	
other	Yes	`libmkl_sequential.a`	None	
other	No	`libmkl_intel_thread.a`	`libguide.so` or `libiomp5.so`	

NOTE. If you compiled your application with `libguide` from Intel MKL 9.x or earlier, then the you cannot use Intel MKL 10.0 with `libiomp`.

More Linking Examples

Below are some specific examples for linking using Intel® compilers on systems based on Intel® 64 architecture. In these examples, *<MKL path>* and *<MKL include>* placeholders are replaced with user-defined environment variables `$MKLPATH` and `$MKLINCLUDE`, respectively. See also examples on linking with ScaLAPACK and Cluster FFT in chapter 9.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
```

```
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -lguide -lpthread
```

static linking of user code `myprog.f` and parallel Intel MKL supporting LP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -lguide -lpthread
```

dynamic linking of user code `myprog.f` and parallel Intel MKL supporting LP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_sequential.a
$MKLPATH/libmkl_core.a -Wl,--end-group -lpthread
```

static linking of user code `myprog.f` and sequential version of Intel MKL supporting LP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread
```

dynamic linking of user code `myprog.f` and sequential version of Intel MKL supporting LP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -lguide -lpthread
```

static linking of user code `myprog.f` and parallel Intel MKL supporting ILP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_ilp64 -lmkl_intel_thread -lmkl_core -lguide -lpthread
```

dynamic linking of user code `myprog.f` and parallel Intel MKL supporting ILP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_lapack95
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -lguide -lpthread
```

static linking of user code `myprog.f`, Fortran 95 LAPACK interface[1], and parallel Intel MKL supporting LP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_blas95
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -lguide -lpthread
```

1. See section Fortran 95 Interfaces and Wrappers to LAPACK and BLAS in chapter 7 for information on how to build Fortran 95 LAPACK and BLAS interface libraries.

static linking of user code `myprog.f`, Fortran 95 BLAS interface[1], and parallel Intel MKL supporting LP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_solver_lp64.a
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/liblmkl_core.a -Wl,--end-group -lguide -lpthread
```

static linking of user code `myprog.f`, parallel version of sparse solver, and parallel Intel MKL supporting LP64 interface.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_solver_lp64_sequential.a
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_sequential.a
$MKLPATH/libmkl_core.a -Wl,--end-group -lpthread
```

static linking of user code `myprog.f`, sequential version of sparse solver, and sequential Intel MKL supporting LP64 interface.

For other linking examples, see the Intel MKL support website at http://www.intel.com/support/performancetools/libraries/mkl/.

Notes on Linking

Updating LD_LIBRARY_PATH

When using the Intel MKL shared libraries, do not forget to update the shared libraries environment path, that is, a system variable `LD_LIBRARY_PATH`, to include the libraries location. For example, if the Intel MKL libraries are in the `/opt/intel/mkl/10.0.xxx/lib/32` directory (where `xxx` is the Intel MKL package number, for instance, `"039"`), then the following command line can be used (assuming a bash shell):

```
export LD_LIBRARY_PATH=/opt/intel/mkl/10.0.xxx/lib/32:$LD_LIBRARY_PATH
```

Linking with libguide

If you link with `libguide` statically (discouraged)

- and use the Intel® compiler, then link in the `libguide` version that comes with the compiler, that is, use `-openmp` option.
- but do not use the Intel compiler, then link in the `libguide` version that comes with Intel MKL.

If you use dynamic linking (`libguide.so`) of the threading library (recommended), make sure the `LD_LIBRARY_PATH` is defined so that exactly this version of `libguide` is found and used at run time.

Building Custom Shared Objects

Custom shared objects enable reducing the collection of functions available in Intel MKL libraries to those required to solve your particular problems, which helps to save disk space and build your own dynamic libraries for distribution.

Intel MKL Custom Shared Object Builder

Custom shared object builder is targeted for creation of a dynamic library (shared object) with selected functions and located in `tools/builder` directory. The builder contains a makefile and a definition file with the list of functions. The makefile has three targets: "`ia32`", "`ipf`", and "`em64t`". "`ia32`" target is used for processors using IA-32 architecture, "`ipf`" is used for IA-64 architecture, and "`em64t`" is used for Intel® Xeon® processor using Intel® 64 architecture.

Specifying Makefile Parameters

There are several macros (parameters) for the makefile:

`export = functions_list`

determines the name of the file that contains the list of entry point functions to be included into shared object. This file is used for definition file creation and then for export table creation. Default name is `functions_list`.

`name = mkl_custom`

specifies the name of the created library. By default, the library `mkl_custom.so` is built.

`xerbla = user_xerbla.o`

specifies the name of object file that contains user's error handler. This error handler will be added to the library and then will be used instead of standard MKL error handler `xerbla`. By default, that is, when this parameter is not specified, standard MKL `xerbla` is used.

All parameters are not mandatory. In the simplest case, the command line could be `make ia32` and the values of the remaining parameters will be taken by default. As a result, `mkl_custom.so` library for processors using IA-32 architecture will be created, the functions list will be taken from the `functions_list.txt` file, and the standard MKL error handler `xerbla` will be used.

Another example for a more complex case is as follows:

```
make ia32 export=my_func_list.txt name=mkl_small xerbla=my_xerbla.o
```

In this case, `mkl_small.so` library for processors using IA-32 architecture will be created, the functions list will be taken from `my_func_list.txt` file, and user's error handler `my_xerbla.o` will be used.

The process is similar for processors using Intel® 64 or IA-64 architecture.

Specifying List of Functions

Entry points in `functions_list` file should be adjusted to interface. For example, Fortran functions get an underscore character `"_"` as a suffix when added to the library:

```
dgemm_
ddot_
dgetrf_
```

If selected functions have several processor-specific versions, they all will be included into the custom library and managed by dispatcher.

Managing Performance and Memory

6

The chapter features different ways to obtain best performance with Intel® Math Kernel Library (Intel® MKL): primarily, it discusses threading (see Using Intel MKL Parallelism), then shows coding techniques and gives hardware configuration tips for improving performance. The chapter also discusses the Intel MKL memory management and shows how to redefine memory functions that the library uses by default.

Using Intel MKL Parallelism

Intel MKL is threaded in a number of places: direct sparse solver, LAPACK (`*GETRF`, `*POTRF`, `*GBTRF`, `*GEQRF`, `*ORMQR`, `*STEQR`, `*BDSQR`, `*SPTRF`, `*SPTRS`, `*HPTRF`, `*HPTRS`, `*PPTRF`, `*PPTRS` routines), all Level 3 BLAS, Sparse BLAS matrix-vector and matrix-matrix multiply routines for the compressed sparse row and diagonal formats, VML, and all FFTs (except 1D transformations when `DFTI_NUMBER_OF_TRANSFORMS`=1 and sizes are not power of two).

NOTE. For power-of-two data in 1D FFTs, Intel MKL provides parallelism for all the three supported architectures. For Intel® 64 architecture, the parallelism is provided for double complex out-of-place FFTs only.

The library uses OpenMP* threading software, which responds to the environmental variable `OMP_NUM_THREADS` that sets the number of threads to use. Notice that there are different means to set the number of threads. In Intel MKL releases earlier than 10.0, you could use the environment variable `OMP_NUM_THREADS` (see Setting the Number of Threads Using OpenMP Environment Variable for details) or the equivalent OpenMP run-time function calls (detailed in section Changing the Number of Threads at Run Time). Starting with version 10.0, Intel MKL also offers variables that are independent of OpenMP, such as `MKL_NUM_THREADS`, and equivalent Intel MKL functions for threading management (see Using Additional Threading Control for details). The Intel MKL variables are always

inspected first, then the OpenMP variables are examined, and if neither is used, the OpenMP software chooses the default number of threads. This is a change with respect to Intel MKL versions 9.x or earlier, which used a default value of one, as the Intel® Compiler OpenMP software uses the default number of threads equal to the number of processors in your system.

NOTE. In Intel MKL 10.0, OpenMP determines the default number of threads.

It is recommended that you always set `OMP_NUM_THREADS` to the number of processors you wish to use in your application. Do this by any availble means, which are summarized in section Techniques to Set the Number of Threads.

Techniques to Set the Number of Threads

You can employ different techniques to specify the number of threads to use in Intel MKL.

- Set OpenMP or Intel MKL environment variable:
  - `OMP_NUM_THREADS`
  - `MKL_NUM_THREADS`
  - `MKL_DOMAIN_NUM_THREADS`
- Call OpenMP or Intel MKL function:
  - `omp_set_num_threads()`
  - `mkl_set_num_threads()`
  - `mkl_domain_set_num_threads()`.

When choosing the appropriate technique, take into account the following rules:

- If you employ the OpenMP techniques (`OMP_NUM_THREADS` and `omp_set_num_threads()`) only, which was the case with earlier Intel MKL versions, the library will still respond to them.
- The Intel MKL threading controls take precedence over the OpenMP techniques.
- A subroutine call takes precedence over any environment variables. The exception is the OpenMP subroutine `omp_set_num_threads()`, which does not have precedence over Intel MKL environment variables, such as `MKL_NUM_THREADS`.
- The environment variables cannot be used to change run-time behavior in the course of the run, as they are read only once.

Avoiding Conflicts in the Execution Environment

There are situations in which conflicts can exist in the execution environment that make the use of threads in Intel MKL problematic. They are listed here with recommendations for dealing with these. First, a brief discussion of why the problem exists is appropriate.

If the user threads the program using OpenMP directives and compiles the program with Intel® compilers, Intel MKL and the program will both use the same threading library. Intel MKL tries to determine if it is in a parallel region in the program, and if it is, it does not spread its operations over multiple threads unless the user specifically requests Intel MKL to do so via the `MKL_DYNAMIC` functionality (see Using Additional Threading Control for details). However, Intel MKL can be aware that it is in a parallel region only if the threaded program and Intel MKL are using the same threading library. If the user's program is threaded by some other means, Intel MKL may operate in multithreaded mode and the performance may suffer due to overuse of the resources.

Here are several cases with recommendations depending on the threading model you employ:

Table 6-1 How to avoid conflicts in the execution environment for your threading model

Threading model	Discussion
You thread the program using OS threads (`pthreads` on Linux*).	If more than one thread calls the library, and the function being called is threaded, it may be important that you turn off Intel MKL threading. Set the number of threads to one by any of the available means (see Techniques to Set the Number of Threads).
You thread the program using OpenMP directives and/or pragmas and compile the program using a compiler other than a compiler from Intel.	This is more problematic in that setting of `OMP_NUM_THREADS` in the environment affects both the compiler's threading library and `libguide` (`libiomp`). In this case, you should try to choose the Threading layer library that matches the layered Intel MKL with the OpenMP compiler you employ (see Linking with Threading Libraries on how to do this). If this is impossible, the sequential version of Intel MKL can be used as the Threading layer. To do this, you should link with the appropriate Threading layer library: `libmkl_sequential.a` or `libmkl_sequential.so` (see the High-level Directory Structure section in chapter 3).

Table 6-1 How to avoid conflicts in the execution environment for your threading model (continued)

Threading model	Discussion
There are multiple programs running on a multiple-cpu system, as in the case of a parallelized program running using MPI for communication in which each processor is treated as a node.	The threading software will see multiple processors on the system even though each processor has a separate MPI process running on it. In this case, set the number of threads to one by any of the available means (see Techniques to Set the Number of Threads).

To avoid correctness and performance problems, you are also strongly encouraged to dynamically link with the Intel® Legacy OpenMP run-time library `libguide` and Intel® Compatibility OpenMP run-time library `libiomp`.

Setting the Number of Threads Using OpenMP Environment Variable

You can set the number of threads using the environment variable `OMP_NUM_THREADS`. To change the number of threads, in the command shell in which the program is going to run, enter:

`export OMP_NUM_THREADS=<number of threads to use>` for certain shells, such as bash.

or

`set OMP_NUM_THREADS=<number of threads to use>` for other shells, such as csh or tcsh.

See Using Additional Threading Control on how to set the number of threads using Intel MKL environment variables, for example, `MKL_NUM_THREADS`.

Changing the Number of Threads at Run Time

It is not possible to change the number of processors during run time using the environment variables. However, you can call OpenMP API functions from your program to change the number of threads during run time. The following sample code demonstrates changing the number of threads during run time using the `omp_set_num_threads()` routine. See also Techniques to Set the Number of Threads.

To run this example, use the `omp.h` header file from the Intel® Compiler package. If you do not have the Intel Compiler but wish to explore the functionality in the example, use Fortran API for `omp_set_num_threads()` rather than the C version.

Example 6-1 Changing the number of processors for threading

```
#include "omp.h"
#include "mkl.h"
#include <stdio.h>

#define SIZE 1000

void main(int args, char *argv[]){

   double *a, *b, *c;
   a = new double [SIZE*SIZE];
   b = new double [SIZE*SIZE];
   c = new double [SIZE*SIZE];

   double alpha=1, beta=1;
   int m=SIZE, n=SIZE, k=SIZE, lda=SIZE, ldb=SIZE, ldc=SIZE, i=0, j=0;
   char transa='n', transb='n';

   for( i=0; i<SIZE; i++){
       for( j=0; j<SIZE; j++){
           a[i*SIZE+j]= (double)(i+j);
           b[i*SIZE+j]= (double)(i*j);
           c[i*SIZE+j]= (double)0;
       }
   }
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                   m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
```

Example 6-1 Changing the number of processors for threading (continued)

```
printf("row\ta\tc\n");
for ( i=0;i<10;i++){
    printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
}

omp_set_num_threads(1);

for( i=0; i<SIZE; i++){
    for( j=0; j<SIZE; j++){
        a[i*SIZE+j]= (double)(i+j);
        b[i*SIZE+j]= (double)(i*j);
        c[i*SIZE+j]= (double)0;
    }
}
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
               m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

 printf("row\ta\tc\n");
 for ( i=0;i<10;i++){
     printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
 }

 omp_set_num_threads(2);
 for( i=0; i<SIZE; i++){
     for( j=0; j<SIZE; j++){
         a[i*SIZE+j]= (double)(i+j);
         b[i*SIZE+j]= (double)(i*j);
         c[i*SIZE+j]= (double)0;
     }
 }
```

Example 6-1 Changing the number of processors for threading (continued)

```
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                        m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

        printf("row\ta\tc\n");
        for ( i=0;i<10;i++){
            printf("%d:\t%f\t%f\n", i, a[i*SIZE],
    c[i*SIZE]);
        }

        delete [] a;
        delete [] b;
        delete [] c;
    }
```

Using Additional Threading Control

Intel MKL 10.0 introduces new optional threading controls, that is, the environment variables and service functions. They behave similar to their OpenMP equivalents, but take precedence over them. By using these controls along with OpenMP variables, you can thread the part of the application that does not call Intel MKL and the library independently from each other.

These controls enable you to specify the number of threads for Intel MKL independently of the OpenMP settings. Although Intel MKL may actually use the number of threads that differs from the one suggested, the controls will also enable you to instruct the library to try using the suggested number in the event of undetectable threading behavior in the application calling the library.

NOTE. Intel MKL does not always have a choice on the number of threads for certain reasons, such as system resources.

Employing Intel MKL threading controls in your application is optional. If you do not use them, the library will mainly behave the same way as Intel MKL 9.1 in what relates to threading with the possible exception of a different default number of threads. See Note on FFT Usage for the usage differences.

Table 6-2 lists the Intel MKL environment variables for threading control, their equivalent functions, and OMP counterparts:

Table 6-2 Intel MKL environment variables for threading controls

Environment Variable	Service Function	Comment	Equivalent OMP Environment Variable
MKL_NUM_THREADS	mkl_set_num_threads	Suggests the number of threads to use.	OMP_NUM_THREADS
MKL_DOMAIN_NUM_THREADS	mkl_domain_set_num_threads	Suggests the number of threads for a particular function domain.	
MKL_DYNAMIC	mkl_set_dynamic	Enables Intel MKL to dynamically change the number of threads.	OMP_DYNAMIC

NOTE. The functions take precedence over the respective environment variables.
In particular, if in your application, you want Intel MKL to use a given number of threads and do not want users of your application to change this via environment variables, set this number of threads by a call to `mkl_set_num_threads()`, which will have full precedence over any environment variables set.

The example below illustrates the use of the Intel MKL function `mkl_set_num_threads()` to mimic the Intel MKL 9.x default behavior, that is, running on one thread.

Example 6-2 Setting the number of threads to one

```
#include <omp.h>
#include <mkl.h>
…
mkl_set_num_threads ( 1 );
```

The section further expands on the Intel MKL environment variables for threading control. See the *Intel MKL Reference Manual* for the detailed description of the threading control functions, their parameters, calling syntax, and more code examples.

MKL_DYNAMIC

The value of MKL_DYNAMIC is by default set to TRUE, regardless of OMP_DYNAMIC, whose default value may be FALSE.

MKL_DYNAMIC being TRUE means that Intel MKL will always try to pick what it considers the best number of threads, up to the maximum specified by the user. MKL_DYNAMIC being FALSE means that Intel MKL will not deviate from the number of threads the user requested, unless there are reasons why it has no choice.

Notice that setting MKL_DYNAMIC=FALSE does not ensure that Intel MKL will use the number of threads that you request. The library may examine the problem and pick a different number of threads than the value suggested. For example, if you attempt to do a size 1 matrix-matrix multiply across 8 threads, the library may instead choose to use only one thread because it is impractical to use 8 threads in this event.

Note also that if Intel MKL is called in a parallel region, it will use only one thread by default. If you want the library to use nested parallelism, and thread within a parallel region is compiled with the same OpenMP compiler as Intel MKL is using, you may experiment with setting MKL_DYNAMIC to FALSE and manually increasing the number of threads.

In general, you should set MKL_DYNAMIC to FALSE only under circumstances that Intel MKL is unable to detect, for example, when nested parallelism is desired where the library is called already from a parallel section.

MKL_DOMAIN_NUM_THREADS

MKL_DOMAIN_NUM_THREADS accepts a string value *<MKL-env-string>*, which must have the following format:

<MKL-env-string> ::= *<MKL-domain-env-string>* { *<delimiter>* *<MKL-domain-env-string>* }

<delimiter> ::= [*<space-symbol>**] (*<space-symbol>* | *<comma-symbol>* | *<semicolon-symbol>* | *<colon-symbol>*) [*<space-symbol>**]

<MKL-domain-env-string> ::= *<MKL-domain-env-name>* *<uses>* *<number-of-threads>*

<MKL-domain-env-name> ::= MKL_ALL | MKL_BLAS | MKL_FFT | MKL_VML

<uses> ::= [*<space-symbol>**] (*<space-symbol>* | *<equality-sign>* | *<comma-symbol>*) [*<space-symbol>**]

<number-of-threads> ::= *<positive-number>*

<positive-number> ::= *<decimal-positive-number>* | *<octal-number>* | *<hexadecimal-number>*

In the syntax above, MKL_BLAS indicates the BLAS function domain, MKL_FFT indicates non-cluster FFTs, and MKL_VML indicates the Vector Mathematics Library.

For example,

```
MKL_ALL 2 : MKL_BLAS 1 : MKL_FFT 4

MKL_ALL=2 : MKL_BLAS=1 : MKL_FFT=4

MKL_ALL=2,  MKL_BLAS=1,  MKL_FFT=4

MKL_ALL=2;  MKL_BLAS=1;  MKL_FFT=4

MKL_ALL  = 2  MKL_BLAS 1 ,  MKL_FFT  4

MKL_ALL,2: MKL_BLAS 1, MKL_FFT,4 .
```

The global variables MKL_ALL, MKL_BLAS, MKL_FFT, and MKL_VML, as well as the interface for the Intel MKL threading control functions, can be found in the mkl.h header file.

Table 6-3 illustrates how values of MKL_DOMAIN_NUM_THREADS are interpreted.

Table 6-3 Interpretation of MKL_DOMAIN_NUM_THREADS values

Value of MKL_DOMAIN_NUM_THREADS	Interpretation
MKL_ALL=4	All parts of Intel MKL are suggested to try using 4 threads. The actual number of threads may be still different because of the MKL_DYNAMIC setting or system resource issues. The setting is equivalent to MKL_NUM_THREADS = 4.
MKL_ALL=1, MKL_BLAS=4	All parts of Intel MKL are suggested to use 1 thread, except for BLAS, which is suggested to try 4 threads.
MKL_VML = 2	VML is suggested to try 2 threads. The setting affects no other part of Intel MKL.

NOTE. The domain-specific settings take precedence over the overall ones. For example, the "MKL_BLAS=4" value of MKL_DOMAIN_NUM_THREADS suggests to try 4 threads for BLAS, regardless of later setting MKL_NUM_THREADS, and a function call "mkl_domain_set_num_threads (4, MKL_BLAS);" suggests the same, regardless of later calls to mkl_set_num_threads(). However, pay attention to that a function call with input "MKL_ALL", such as "mkl_domain_set_num_threads (4, MKL_ALL);" is equivalent to "mkl_set_num_threads(4)", and thus it will be overwritten by later calls to mkl_set_num_threads. Similarly, the environment setting of MKL_DOMAIN_NUM_THREADS with "MKL_ALL=4" will be overwritten with MKL_NUM_THREADS = 2.

Whereas the MKL_DOMAIN_NUM_THREADS environment variable enables you set several variables at once, for example, "MKL_BLAS=4,MKL_FFT=2", the corresponding function does not take string syntax. So, to do the same with the function calls, you may need to make several calls, which in this example are as follows:

```
mkl_domain_set_num_threads ( 4, MKL_BLAS );
mkl_domain_set_num_threads ( 2, MKL_FFT );
```

Setting the Environment Variables for Threading Control

To set the environment variables used for threading control, in the command shell in which the program is going to run, enter:

`export <VARIABLE NAME>=<value>` for certain shells, such as bash.

For example,

```
export MKL_NUM_THREADS=4
export MKL_DOMAIN_NUM_THREADS="MKL_ALL=1, MKL_BLAS=4"
export MKL_DYNAMIC=FALSE
```

For other shells, such as csh or tcsh, enter

`set <VARIABLE NAME>=<value> .`

For example,

```
set MKL_NUM_THREADS=4
set MKL_DOMAIN_NUM_THREADS="MKL_ALL=1, MKL_BLAS=4"
set MKL_DYNAMIC=FALSE
```

Note on FFT Usage

Introduction of additional threading control made it possible to optimize the *commit* stage of the FFT implementation and get rid of double data initialization. However, this optimization requires a change in the FFT usage. Suppose you create threads in the application yourself after initializing all FFT descriptors. In this case, threading is employed for the parallel FFT computation only, the descriptors are released upon return from the parallel region, and each descriptor is used only within the corresponding thread. Starting with Intel MKL 10.0, you must explicitly instruct the library before the *commit* stage to work on one thread. To do this, set `MKL_NUM_THREADS=1` or `MKL_DOMAIN_NUM_THREADS="MKL_FFT=1"` or call the corresponding pair of service functions. Otherwise, the actual number of threads may be different because the `DftiCommitDescriptor` function is not in a parallel region. See *Example C-27a "Using Parallel Mode with Multiple Descriptors Initialized in One Thread"* in the *Intel MKL Reference Manual*.

Tips and Techniques to Improve Performance

To obtain the best performance with Intel MKL, follow the recommendations given in the subsections below.

Coding Techniques

To obtain the best performance with Intel MKL, ensure the following data alignment in your source code:

- arrays are aligned at 16-byte boundaries
- leading dimension values (`n*element_size`) of two-dimensional arrays are divisible by 16
- for two-dimensional arrays, leading dimension values divisible by 2048 are avoided.

LAPACK packed routines

The routines with the names that contain the letters `HP`, `OP`, `PP`, `SP`, `TP`, `UP` in the matrix type and storage position (the second and third letters respectively) operate on the matrices in the packed format (see LAPACK "*Routine Naming Conventions*" sections in the *Intel MKL Reference Manual*). Their functionality is strictly equivalent to the functionality of the unpacked routines with the names containing the letters `HE`, `OR`, `PO`, `SY`, `TR`, `UN` in the corresponding positions, but the performance is significantly lower.

If the memory restriction is not too tight, use an unpacked routine for better performance. Note that in such a case, you need to allocate $N^2/2$ more memory than the memory required by a respective packed routine, where N is the problem size (the number of equations).

For example, solving a symmetric eigenproblem with an expert driver can be speeded up through using an unpacked routine:

```
call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w,
z, ldz, work, lwork, iwork, ifail, info),
```

where *a* is the dimension *lda*-by-*n*, which is at least N^2 elements, instead of

```
call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z,
ldz, work, iwork, ifail, info),
```

where *ap* is the dimension *N**(*N*+1)/2.

FFT functions

There are additional conditions to gain performance of the FFT functions.

Applications based on IA-32 or Intel® 64 architecture. The addresses of the first elements of arrays and the leading dimension values, in bytes (`n*element_size`), of two-dimensional arrays should be divisible by cache line size, which equals

- 32 bytes for Pentium® III processor
- 64 bytes for Pentium® 4 processor
- 128 bytes for processor using Intel® 64 architecture.

Applications based on IA-64 architecture. The sufficient conditions are as follows:

- For the C-style FFT, the distance *L* between arrays that represent real and imaginary parts is not divisible by 64. The best case is when *L*=*k**64 + 16
- Leading dimension values, in bytes (`n*element_size`), of two-dimensional arrays are not power of two.

Hardware Configuration Tips

Dual-Core Intel® Xeon® processor 5100 series systems. To get the best Intel MKL performance on Dual-Core Intel® Xeon® processor 5100 series systems, you are advised to enable the *Hardware DPL (streaming data) Prefetcher* functionality of this processor. Configuration of this functionality is accomplished through appropriate BIOS settings where supported. Check your BIOS documentation for details.

The use of Hyper-Threading Technology. Hyper-Threading Technology (HT Technology) is especially effective when each thread is performing different types of operations and when there are under-utilized resources on the processor. Intel MKL fits neither of these criteria as the threaded portions of the library execute at high efficiencies using most of the available resources and perform identical operations on each thread. You may obtain higher performance when using Intel MKL *without HT Technology enabled*.

Managing Multi-core Performance

You can obtain best performance on systems with multi-core processors by requiring that threads do not migrate from core to core. To do this, bind threads to the CPU cores by setting an affinity mask to threads. You can do it either with OpenMP facilities (which is recommended if available, for instance, via `KMP_AFFINITY` environment variable using Intel OpenMP), or with a system routine, as in the example below.

Suppose,

- The system has two sockets with two cores each
- 2 threads parallel application, which calls Intel MKL FFT, happens to run faster than in 4 threads, but the performance in 2 threads is very unstable

In this case,

1. Put the part of the following code fragment preceding the last comment into your code before FFT call to bind the threads to the cores on different sockets.
2. Build your application and run it in 2 threads:

   ```
   env OMP_NUM_THREADS=2 ./a.out
   ```

Example 6-3 Setting an affinity mask by operating system means using Intel® compiler

```
// Set affinity mask
#include <sched.h>
#include <omp.h>
#pragma omp parallel default(shared)
{
   unsigned long mask = (1 << omp_get_thread_num()) * 2;
   sched_setaffinity( 0, sizeof(mask), &mask );
     }
// Call MKL FFT routine
```

See the *Linux Programmer's Manual* (in man pages format) for particulars of the `sched_setaffinity` function used in the above example.

Operating on Denormals

If an Intel MKL function operates on denormals, that is, non-zero numbers that are smaller than the smallest possible non-zero number supported by a given floating-point format, or produces denormals during the computation (for instance, if the incoming data is too close to the underflow threshold), you may experience considerable performance drop. The CPU state may be set so that floating-point operations on denormals invoke the exception handler that slows down the application.

To resolve the issue, before compiling the main program, turn on the `-ftz` option, if you are using the Intel® compiler or any other compiler that can control this feature. In this case, denormals are treated as zeros at processor level and the exception handler is not invoked. Note, however, that setting this option slightly impacts the accuracy.

Another way to bring the performance back to norm is proper scaling of the input data to avoid numbers near the underflow threshold.

FFT Optimized Radices

You can gain performance of Intel MKL FFT if length of the data vector permits factorization into powers of optimized radices.

In Intel MKL, the list of optimized radices depends upon the architecture:

- 2, 3, 4, 5 for IA-32 architecture
- 2, 3, 4, 5 for Intel® 64 architecture
- 2, 3, 4, 5, 7, 11 for IA-64 architecture.

Using Intel MKL Memory Management

Intel MKL has memory management software that controls memory buffers for use by the library functions. New buffers that the library allocates when certain functions (Level 3 BLAS or FFT) are called are not deallocated until the program ends. If at some point your program needs to free memory, it may do so with a call to `MKL_FreeBuffers()`. If another call is made to a library function that needs a memory buffer, then the memory manager will again allocate the buffers and they will again remain allocated until either the program ends or the program deallocates the memory.

This behavior facilitates better performance. However, some tools may report the behavior as a memory leak. You can release memory in your program through the use of a function made available in Intel MKL or you can force memory releasing after each call by setting an environment variable.

The memory management software is turned on by default. To disable the software using the environment variable, set `MKL_DISABLE_FAST_MM` to any value, which will cause memory to be allocated and freed from call to call. Disabling this feature will negatively impact performance of routines such as the level 3 BLAS, especially for small problem sizes.

Using one of these methods to release memory will not necessarily stop programs from reporting memory leaks, and, in fact, may increase the number of such reports in case you make multiple calls to the library, thereby requiring new allocations with each call. Memory not released by one of the methods described will be released by the system when the program ends.

Memory management has a restriction for the number of allocated buffers in each thread. Currently this number is 32. The maximum number of supported threads is 514. To avoid the default restriction, disable memory management.

Redefining Memory Functions

Starting with MKL 9.0, you can replace memory functions that the library uses by default with your own ones. It is possible due to the *memory renaming* feature.

Memory renaming

In general, if users try to employ their own memory management functions instead of similar system functions (`malloc`, `free`, `calloc`, and `realloc`), actually, the memory gets managed by two independent memory management packages, which may cause memory issues. To prevent from such issues, the memory renaming feature was introduced in certain Intel® libraries and in particular in Intel MKL. This feature enables users to redefine memory management functions.

Redefining is possible because Intel MKL actually uses pointers to memory functions (`i_malloc`, `i_free`, `i_calloc`, `i_realloc`) rather than the functions themselves. These pointers initially hold addresses of respective system memory management functions (`malloc`, `free`, `calloc`, `realloc`) and are visible at the application level. So, the pointer values can be redefined programmatically.

Once a user has redirected these pointers to their own respective memory management functions, the memory will be managed with user-defined functions rather than system ones. As only one (user-defined) memory management package is in operation, the issues are avoided.

Intel MKL memory management by default uses standard C run-time memory functions to allocate or free memory. These functions can be replaced using memory renaming.

How to redefine memory functions

To redefine memory functions, you may use the following procedure:

1. Include the `i_malloc.h` header file in your code.
   (The header file contains all declarations required for an application developer to replace the memory allocation functions. This header file also describes how memory allocation can be replaced in those Intel libraries that support this feature.)
2. Redefine values of pointers `i_malloc`, `i_free`, `i_calloc`, `i_realloc` prior to the first call to MKL functions:

Example 6-4 Redefining memory functions

```
#include "i_malloc.h"
   . . .
   i_malloc  = my_malloc;
   i_calloc  = my_calloc;
   i_realloc = my_realloc;
   i_free    = my_free;
   . . .
// Now you may call Intel MKL functions
```

Language-specific Usage Options

7

Intel® Math Kernel Library (Intel® MKL) basically provides support for Fortran and C/C++ programming. However, not all function domains support both Fortran and C interfaces (see Table A-1). For example, LAPACK has no C interface. Still you can call functions comprising these domains from C using mixed-language programming.

Moreover, even if you want to use LAPACK or BLAS, which basically support Fortran, in the Fortran 95 environment, additional effort is initially required to build language-specific interface libraries and modules, being delivered as source code.

The chapter mainly focuses on mixed-language programming and the use of language-specific interfaces. It expands upon the use of Intel MKL in C language environments for function domains that basically support Fortran as well as explains usage of language-specific interfaces and, in particular, Fortran 95 interfaces to LAPACK and BLAS. In this connection, compiler-dependent functions are discussed to explain why Fortran 95 modules are supplied as sources. A separate section guides you through the process of running examples of invoking Intel MKL functions from Java.

Using Language-Specific Interfaces with Intel MKL

The following interface libraries and modules may be generated as a result of operation of respective makefiles located in the interfaces directory.

Table 7-1 Interface libraries and modules

File name	Comment
`libmkl_blas95.a`	Contains Fortran 95 wrappers for BLAS (BLAS95)
`libmkl_lapack95.a`	Contains Fortran 95 wrappers for LAPACK (LAPACK95)
`libfftw2xc_intel.a`	Contains interfaces for FFTW version 2.x (C interface for Intel® compiler) to call Intel MKL FFTs.
`libfftw2xc_gnu.a`	Contains interfaces for FFTW version 2.x (C interface for GNU compiler) to call Intel MKL FFTs.

Table 7-1 Interface libraries and modules (continued)

File name	Comment
libfftw2xf_intel.a	Contains interfaces for FFTW version 2.x (Fortran interface for Intel compiler) to call Intel MKL FFTs.
libfftw2xf_gnu.a	Contains interfaces for FFTW version 2.x (Fortran interface for GNU compiler) to call Intel MKL FFTs.
libfftw3xc_intel.a	Contains interfaces for FFTW version 3.x (C interface for Intel compiler) to call Intel MKL FFTs.
libfftw3xc_gnu.a	Contains interfaces for FFTW version 3.x (C interface for GNU compiler) to call Intel MKL FFTs.
libfftw3xf_intel.a	Contains interfaces for FFTW version 3.x (Fortran interface for Intel compiler) to call Intel MKL FFTs.
libfftw3xf_gnu.a	Contains interfaces for FFTW version 3.x (Fortran interface for GNU compiler) to call Intel MKL FFTs.
libfftw2x_cdft_SINGLE.a	Contains single-precision interfaces for MPI FFTW version 2.x (C interface) to call Intel MKL cluster FFTs.
libfftw2x_cdft_DOUBLE.a	Contains double-precision interfaces for MPI FFTW version 2.x (C interface) to call Intel MKL cluster FFTs.
mkl95_blas.mod	Contains Fortran 95 interface module for BLAS (BLAS95)
mkl95_lapack.mod	Contains Fortran 95 interface module for LAPACK (LAPACK95)
mkl95_precision.mod	Contains Fortran 95 definition of precision parameters for BLAS95 and LAPACK95

Section "Fortran 95 Interfaces and Wrappers to LAPACK and BLAS" shows by example how these libraries and modules are generated.

Fortran 95 Interfaces and Wrappers to LAPACK and BLAS

Fortran 95 interfaces are provided for pure procedures and along with wrappers are delivered as sources. (For more information, see Compiler-dependent Functions and Fortran 95 Modules). The simplest way to use them is building corresponding libraries and linking them as user's libraries. To do this, you must have administrator rights. Provided the product directory is open for writing, the procedure is simple:

1. Go to the respective directory `mkl/10.0.xxx/interfaces/blas95` or `mkl/10.0.xxx/interfaces/lapack95`
   where `xxx` is the Intel MKL package number, for example, `"039"`
2. Type one of the following commands:
   `make PLAT=lnx32 lib` - for IA-32 architecture
   `make PLAT=lnx32e lib` - for Intel® 64 architecture
   `make PLAT=lnx64 lib` - for IA-64 architecture.

As a result, the required library and a respective .mod file will be built and installed in the standard catalog of the release.

The .mod files can also be obtained from files of interfaces using the compiler command

ifort -c mkl_lapack.f90 or ifort -c mkl_blas.f90.

These files are in the *include* directory.

If you do not have administrator rights, then do the following:

1. Copy the entire directory (mkl/10.0.xxx/interfaces/blas95 or mkl/10.0.xxx/interfaces/lapack95) into a user-defined directory *<user_dir>*
2. Copy the corresponding file (mkl_blas.f90 or mkl_lapack.f90) from mkl/10.0.xxx/include into the user-defined directory *<user_dir>*/blas95 or *<user_dir>*/lapack95 respectively
3. Run one of the above make commands in *<user_dir>*/blas95 or *<user_dir>*/lapack95 with an additional variable, for instance:

   ```
   make PLAT=lnx32 INTERFACE=mkl_blas.f90 lib
   make PLAT=lnx32 INTERFACE=mkl_lapack.f90 lib
   ```

Now the required library and the .mod file will be built and installed in the *<user_dir>*/blas95 or *<user_dir>*/lapack95 directory, respectively.

By default, the ifort compiler is assumed. You may change it with an additional parameter of make: FC=*<compiler>*.

For instance,

```
make PLAT=lnx64 FC=<compiler> lib
```

There is also a way to use the interfaces without building the libraries.

To delete the library from the building directory, use the following commands:

```
make PLAT=lnx32 clean      - for IA-32 architecture
make PLAT=lnx32e clean     - for Intel® 64 architecture
make PLAT=lnx64 clean      - for IA-64 architecture.
```

Compiler-dependent Functions and Fortran 95 Modules

Compiler-dependent functions arise whenever the compiler places into the object code function calls that are resolved in its run-time library (RTL). Linking of such code without the appropriate RTL will result in undefined symbols. MKL has been designed to minimize RTL dependencies.

Where the dependencies do arise, supporting RTL is shipped with Intel MKL. The only example of such RTLs, except those that are relevant to the Intel MKL cluster software, are libguide and libiomp, which are the libraries for the OpenMP* code compiled with an Intel® compiler. libguide and libiomp support the threaded code in Intel MKL.

In other cases where RTL dependencies might arise, the functions are delivered as source code and it is the responsibility of the user to compile the code with whatever compiler employed.

In particular, Fortran 95 modules result in the compiler-specific code generation requiring RTL support, so, Intel MKL delivers these modules as source code.

Mixed-language programming with Intel MKL

Appendix A lists the programming languages supported for each Intel MKL function domain. However, you can call Intel MKL routines from different language environments. This section explains how to do this using mixed-language programming.

Calling LAPACK, BLAS, and CBLAS Routines from C Language Environments

Not all Intel MKL function domains support both C and Fortran environments. To use Intel MKL Fortran-style functions in C/C++ environments, you should observe certain conventions, which are discussed for LAPACK and BLAS in the subsections below.

LAPACK

As LAPACK routines are Fortran-style, when calling them from C-language programs, make sure that you follow the Fortran-style calling conventions:

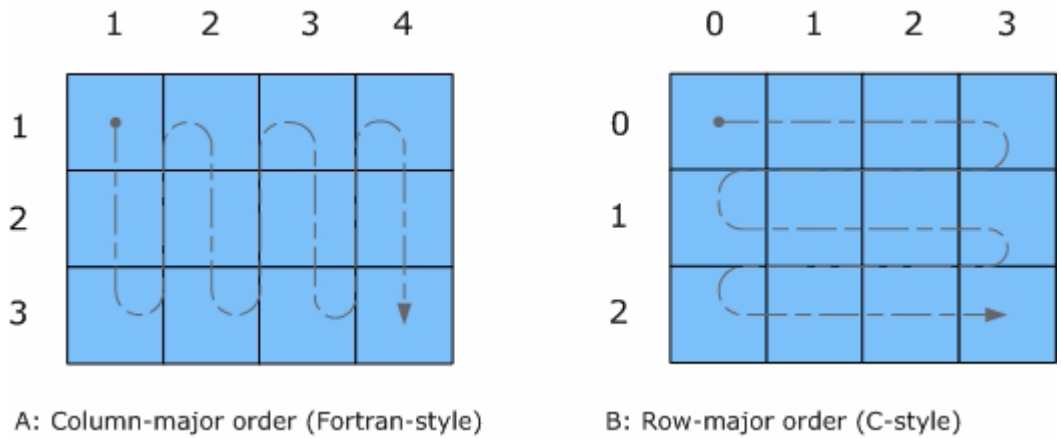
- Pass variables by 'address' as opposed to pass by 'value'.

  Function calls is Example 7-1 and Example 7-2 illustrate this.

- Store your data Fortran-style, that is, in column-major rather than row-major order.

  With row-major order, adopted in C, the last array index changes most quickly and the first one changes most slowly when traversing the memory segment where the array is stored. With Fortran-style column-major order, the last index changes most slowly whereas the first one changes most quickly (as illustrated by Figure 7-1 for a 2D array).

Figure 7-1 Column-major order vs. row-major order



For example, if a two-dimensional matrix A of size `m x n` is stored densely in a one-dimensional array B, you can access a matrix element like this:

`A[i][j] = B[i*n+j]` in C (`i=0, ... , m-1, j=0, ... , n-1`)

`A(i,j) = B(j*m+i)` in Fortran (`i=1, ... , m, j=1, ... , n`).

When calling LAPACK routines from C, also mind that LAPACK routine names can be both upper-case or lower-case, with trailing underscore or not. For example, these names are equivalent: `dgetrf`, `DGETRF`, `dgetrf_`, `DGETRF_`.

BLAS

BLAS routines are Fortran-style routines. If you call BLAS routines from a C-language program, you must follow the Fortran-style calling conventions:

- Pass variables by address as opposed to passing by value.
- Store data Fortran-style, that is, in column-major rather than row-major order.

Refer to the LAPACK section for details of these conventions. See Example 7-1 on how to call BLAS routines from C.

When calling BLAS routines from C, also mind that BLAS routine names can be both upper-case and lower-case, with trailing underscore or not. For example, these names are equivalent: `dgemm`, `DGEMM`, `dgemm_`, `DGEMM_`.

CBLAS

An alternative for calling BLAS routines from a C-language program is to use the CBLAS interface.

CBLAS is a C-style interface to the BLAS routines. You can call CBLAS routines using regular C-style calls. When using the CBLAS interface, the header file `mkl.h` will simplify the program development as it specifies enumerated values as well as prototypes of all the functions. The header determines if the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation. Example 7-3 illustrates the use of CBLAS interface.

Calling BLAS Functions That Return the Complex Values in C/C++ Code

You must be careful when handling a call from C to a BLAS function that returns complex values. The problem arises because these are Fortran functions and complex return values are handled quite differently for C and Fortran. However, in addition to normal function calls, Fortran enables calling functions as though they were subroutines, which provides a mechanism for returning the complex value correctly when the function is called from a C program. When a Fortran function is called as a subroutine, the return value shows up as the first parameter in the calling sequence. This feature can be exploited by the C programmer.

The following example shows how this works.

Normal Fortran function call:	`result = cdotc( n, x, 1, y, 1 ).`
A call to the function as a subroutine:	`call cdotc( result, n, x, 1, y, 1).`
A call to the function from C (notice that the hidden parameter gets exposed):	`cdotc( &result, &n, x, &one, y, &one ).`

NOTE. Intel MKL has both upper-case and lower-case entry points in BLAS, with trailing underscore or not. So, all these names are acceptable: `cdotc`, `CDOTC`, `cdotc_`, `CDOTC_`.

Using the above example, you can call from C, and thus, from C++, several level 1 BLAS functions that return complex values. However, it is still easier to use the CBLAS interface. For instance, you can call the same function using the CBLAS interface as follows:

```
cblas_cdotu( n, x, 1, y, 1, &result )
```

NOTE. The complex value comes back expressly in this case.

The following example illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

In this example, the complex dot product is returned in the structure `c`.

Example 7-1 Calling a complex BLAS Level 1 function from C

```
#include "mkl.h"
#define N 5
void main()
{
int n, inca = 1, incb = 1, i;
typedef struct{ double re; double im; } complex16;
complex16 a[N], b[N], c;
void zdotc();
n = N;

for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
zdotc( &c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

Below is the C++ implementation:

Example 7-2 Calling a complex BLAS Level 1 function from C++

```
#include "mkl.h"
typedef struct{ double re; double im; } complex16;
extern "C" void zdotc (complex16*, int *, complex16 *, int *, complex16
*, int *);

#define N 5

void main()
{
int n, inca = 1, incb = 1, i;

complex16 a[N], b[N], c;

n = N;

for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
zdotc(&c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

The implementation below uses CBLAS:

Example 7-3 Using CBLAS interface instead of calling BLAS directly from C

```
#include "mkl.h"
typedef struct{ double re; double im; } complex16;

extern "C" void cblas_zdotc_sub ( const int , const complex16 *,
    const int , const complex16 *, const int, const complex16*);

#define N 5

void main()
{

int n, inca = 1, incb = 1, i;

complex16 a[N], b[N], c;
n = N;
for( i = 0; i < n; i++ ){

a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
cblas_zdotc_sub(n, a, inca, b, incb,&c );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

Invoking Intel MKL Functions from Java Applications

This section describes examples that are provided with the Intel MKL package and illustrate calling the library functions from Java.

Intel MKL Java examples

Java was positioned by its inventor, the Sun Microsystems Corporation as "Write Once Run Anywhere" (WORA) language. Intel MKL may help to speed-up Java applications, the WORA philosophy being partially supported, as Intel MKL editions are intended for wide variety of operating systems and processors covering most kinds of laptops and desktops, many workstations and servers.

To demonstrate binding with Java, Intel MKL includes the set of Java examples found in the following directory:

`<mkl directory>/examples/java .`

The examples are provided for the following MKL functions:

- the `?gemm`, `?gemv`, and `?dot` families from CBLAS
- complete set of non-cluster FFT functions
- ESSL[1]-like functions for 1-dimensional convolution and correlation.
- VSL Random Number Generators (RNG), except user-defined ones and file subroutines.
- VML functions, except `GetErrorCallBack`, `SetErrorCallBack`, and `ClearErrorCallBack`.

You can see the example sources in the following directory:

`<mkl directory>/examples/java/examples .`

The examples are written in Java. They demonstrate usage of the MKL functions with the following variety of data:

- 1- and 2-dimensional data sequences
- real and complex types of the data
- single and double precision.

However, note that the wrappers, used in examples, do not

- demonstrate the use of huge arrays (>2 billion elements)
- demonstrate processing of arrays in native memory
- check correctness of function parameters
- demonstrate performance optimizations

To bind with Intel MKL, the examples use the Java Native Interface (JNI). The JNI documentation to start with is available from http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html .

1. IBM ESSL* library.

The Java example set includes JNI wrappers which perform the binding. The wrappers do not depend on the examples and may be used in your Java applications. The wrappers for CBLAS, FFT, VML, VSL RNG, and ESSL-like convolution and correlation functions do not depend on each other.

To build the wrappers, just run the examples (see the Running the examples section for details). The makefile builds the wrapper binaries and the examples, invoked after that, double-check if the wrappers are built correctly. As a result of running the examples, the following directories will be created in `<mkl directory>/examples/java`:

- `docs`
- `include`
- `classes`
- `bin`
- `_results` .

The directories `docs`, `include`, `classes`, and `bin` will contain the wrapper binaries and documentation; the directory `_results` will contain the testing results.

For a Java programmer, the wrappers look like the following Java classes:

- `com.intel.mkl.CBLAS`
- `com.intel.mkl.DFTI`
- `com.intel.mkl.ESSL`
- `com.intel.mkl.VML`
- `com.intel.mkl.VSL`

Documentation for the particular wrapper and example classes will be generated from the Java sources during building and running the examples. To browse the documentation, start from the index file in the `docs` directory which will be created by the build script:

`<mkl directory>/examples/java/docs/index.html` .

The Java wrappers for CBLAS, VML, VSL RNG, and FFT establish the interface that directly corresponds to the underlying native functions and you can refer to the Intel MKL Reference Manual for their functionality and parameters. Interfaces for the ESSL-like functions are described in the generated documentation for the `com.intel.mkl.ESSL` class.

Each wrapper consists of the interface part for Java and JNI stub written in C. You can find the sources in the following directory:

`<mkl directory>/examples/java/wrappers` .

Both Java and C parts of the wrapper for CBLAS and VML demonstrate the straightforward approach, which you may easily employ to cover missing CBLAS functions.

The wrapper for FFT is more complicated because of supporting the lifecycle for FFT descriptor objects. To compute a single Fourier transform, an application needs to call the FFT software several times with the same copy of native FFT descriptor. The wrapper provides the handler class to hold the native descriptor while virtual machine runs Java bytecode.

The wrapper for VSL RNG is similar to the one for FFT. The wrapper provides the handler class to hold the native descriptor of the stream state.

The wrapper for the convolution and correlation functions mitigates the same difficulty of the VSL interface, which assumes similar lifecycle for "task descriptors". The wrapper utilizes the ESSL-like interface for those functions, which is simpler for the case of 1-dimensional data. The JNI stub additionally enwraps the MKL functions into the ESSL-like wrappers written in C and so "packs" the lifecycle of a task descriptor into a single call to the native method.

The wrappers meet the JNI Specification versions 1.1 and 5.0 and so must work with virtually every modern implementation of Java.

The examples and the Java part of the wrappers are written for the Java language described in "*The Java Language Specification (First Edition)*" and extended with the feature of "inner classes" (this refers to late 1990s). This level of language version is supported by all versions of Sun's Java Software Development Kit (SDK) and compatible implementations starting from the version 1.1.5, that is, by all modern versions of Java.

The level of C language is "Standard C" (that is, C89) with additional assumptions about integer and floating-point data types required by the Intel MKL interfaces and the JNI header files. That is, the native `float` and `double` data types are required to be the same as JNI `jfloat` and `jdouble` data types, respectively, and the native `int` is required to be 4-byte long.

Running the examples

The Java examples support all the C and C++ compilers that the Intel MKL does. The makefile intended to run the examples also needs the make utility, which is typically provided with the Linux* operating system.

To run Java examples, Java SDK is required for compiling and running Java code. A Java implementation must be installed on the computer or available via the network. You may download the SDK from the vendor website.

The examples must work for all versions of Java 2 SE SDK. However, they were tested only with the following Java implementations:

- from the Sun Microsystems Corporation (http://sun.com)
- from the BEA (http://bea.com)

See the Intel MKL Release Notes about the supported versions of these Java SDKs.

NOTE. The implementation from the Sun Microsystems Corporation supports only processors using IA-32 and Intel® 64 architectures. The implementation from BEA supports Intel® Itanium® 2 processors as well.

Also note that Java Run-time Environment (JRE), which may be pre-installed on your computer, is not enough. You need JDK that supports the following set of tools:

- java
- javac
- javah
- javadoc

To make these tools available for the examples makefile, you have to setup the `JAVA_HOME` environment variable and to add JDK binaries directory to the system `PATH`, for example:

```
export JAVA_HOME=/home/<user name>/jdk1.5.0_09
```

```
export PATH=${JAVA_HOME}/bin:${PATH}
```

You may also need to clear the `JDK_HOME` environment variable, if it is assigned a value:

```
unset JDK_HOME
```

To start the examples, use the makefile found in the Intel MKL Java examples directory:

```
make {so32|soem64t|so64} [function=…] [compiler=…]
```

If started without specifying a target (any of the choices, like `so32`), the makefile prints the help info, which explains the targets as well as the *`function`* and *`compiler`* parameters.

For the examples list, see the `examples.lst` file in the same directory.

Known limitations

There are three kinds of limitations:

- functionality
- performance
- known bugs.

Functionality. It is possible that some MKL functions will not work fine if called from Java environment via a wrapper, like those provided with the Intel MKL Java examples. Only those specific CBLAS, FFT, VML, VSL RNG, and the convolution/correlation functions listed

in the Intel MKL Java examples section were tested with Java environment. So, you may use the Java wrappers for these CBLAS, FFT, VML, VSL RNG, and convolution/correlation functions in your Java applications.

Performance. The functions from Intel MKL must work faster than similar functions written in pure Java. However, note that performance was not the main goal for these wrappers. — The intent was giving code examples. So, an Intel MKL function called from Java application will probably work slower than the same function called from a program written in C/C++ or Fortran.

Known bugs. There is a number of known bugs in Intel MKL (identified in the Release Notes) and there are incompatibilities between different versions of Java SDK. The examples and wrappers include workarounds for these problems to make the examples work anyway. Source codes of the examples and wrappers include comments which describe the workarounds.

Coding Tips 8

This is another chapter whose contents discusses programming with Intel® Math Kernel Library (Intel® MKL). Whereas chapter 7 focuses on general language-specific programming options, this one presents coding tips that may be helpful to meet certain specific needs. Currently the only tip advising how to achieve numerical stability is given. You can find other coding tips, relevant to performance and memory management, in chapter 6.

Aligning Data for Numerical Stability

If linear algebra routines (LAPACK, BLAS) are applied to inputs that are bit-for-bit identical but the arrays are differently aligned or the computations are performed either on different platforms or with different numbers of threads, the outputs may not be bit-for-bit identical, though they will deviate within the appropriate error bounds. The Intel MKL version may also affect numerical stability of the output, as the routines may be implemented differently in different versions. With a given Intel MKL version, the outputs will be bit-for-bit identical provided all the following conditions are met:

- the outputs are obtained on the same platform;
- the inputs are bit-for-bit identical;
- the input arrays are aligned identically at 16-byte boundaries.

Unlike the first two conditions, which are under users' control, the alignment of arrays, by default, is not. For instance, arrays dynamically allocated using `malloc` are aligned at 8-byte boundaries, but not at 16-byte. If you need the numerically stable output, to get the correctly aligned addresses, you may use the appropriate code fragment below:

Example 8-1 Aligning addresses at 16-byte boundaries

```
// C language
...
#include <stdlib.h>
...
void *ptr, *ptr_aligned;
ptr = malloc( sizeof(double)*workspace + 16 );
ptr_aligned = ( (size_t)ptr%16 ? (void*)
((size_t)ptr+16-((size_t)ptr%16)) : ptr );
// call the program using MKL and passing the address aligned to 16-bit
boundary
mkl_app( ptr_aligned );
...
free( ptr );


! Fortran language
...
double precision darray( workspace+1 )
! call the program using MKL and passing the address aligned to 16-bit
boundary
if( mod( %loc(darray), 16 ).eq.0 ) then
     call mkl_app( darray(1) )
else
     call mkl_app( darray(2) )
end if
```

Working with Intel® Math Kernel Library Cluster Software

9

This chapter discusses usage of Intel® MKL ScaLAPACK and Cluster FFTs, mainly describing linking your application with these domains and including C- and Fortran-specific linking examples; gives information on the supported MPI.

See Table 3-7 for detailed Intel MKL directory structure in chapter 3.

For information on the available documentation and the `doc` directory, see Table 3-8 in the same chapter.

For information on MP LINPACK Benchmark for Clusters, see section Intel® Optimized MP LINPACK Benchmark for Clusters in chapter 10.

Intel MKL ScaLAPACK and FFTs support MPICH-1.2.x and Intel® MPI.

To link a program that calls ScaLAPACK, you need to know how to link an MPI application first.

Typically, this involves using *mpi* scripts *mpicc* or *mpif77* (C or Fortran77 scripts) that use the correct MPI header files and others. If, for instance, you are using MPICH installed in `/opt/mpich`, then typically `/opt/mpich/bin/mpicc` and `/opt/mpich/bin/mpif77` will be the compiler scripts and `/opt/mpich/lib/libmpich.a` will be the library used for that installation.

Linking with ScaLAPACK and Cluster FFTs

To link to ScaLAPACK and/or Cluster FFTs in Intel MKL, use the following general form:

```
<<MPI> linker script> <files to link>                    \
        -L<Cluster MKL path> <Cluster MKL Library>  \
        <BLACS> <MKL Core Libraries>                      \
```

where

<MPI> is one of several MPI implementations (MPICH, Intel MPI 1.x, Intel MPI 2.x, Intel MPI 3.x)

<BLACS> is one of `-lmkl_blacs, -lmkl_blacs_intelmpi, -lmkl_blacs_intelmpi20,-mkl_blacs_openmpi`

<MKL Cluster Library> is `-lmkl_scalapack-core` and/or `-lmkl_cdft_core`

<MKL Core Libraries> is *<MKL LAPACK & MKL kernel libraries>* for ScaLAPACK, and *<MKL kernel libraries>* for Cluster FFTs.

<MKL LAPACK & kernel libraries> are LAPACK, processor optimized kernels, threading library, and system library for threading support linked as described at the beginning of section Link Command Syntax in Chapter 5.

Note that *<<MPI> linker script>* and *<BLACS>* library should correspond to the MPI version. For instance, if it is Intel MPI 2.x, then *<Intel MPI 2.x linker script>* and `libmkl_blacs_intelmpi20` libraries are used. To link with Intel MPI 3.0, also `libmkl_blacs_intelmpi20` should be used.

For information on linking with Intel® MKL libraries, see Chapter 5 Linking Your Application with Intel® Math Kernel Library.

Setting the Number of Threads

The OpenMP* software responds to the environmental variable `OMP_NUM_THREADS`. Intel® MKL 10.0 has also introduced other mechanisms to set the number of threads, such as `MKL_NUM_THREADS` or `MKL_DOMAIN_NUM_THREADS` (see section "Using Additional Threading Control" in chapter 6). Make certain that the relevant environment variable has the same and correct value on all the nodes. Intel MKL 10.0 also no longer sets the default number of threads to 1, but depends on the compiler to set the default number. For the threading layer based on the Intel® compiler (`libmkl_intel_thread.a`), this value is the number of CPUs according to the OS. Be cautious to avoid over-prescribing the number of threads, which may occur, for instance, when the number of MPI ranks per node and the number of threads per node are both greater than one.

The best way to set, for example, the environment variable `OMP_NUM_THREADS` is in the login environment. Remember that `mpirun` starts a fresh default shell on all of the nodes and so, changing this value on the head node and then doing the run (which works on an SMP system) will not effectively change the variable as far as your program is concerned. In `.bashrc`, you could add a line at the top, which looks like this:

```
OMP_NUM_THREADS=1; export OMP_NUM_THREADS
```

It is possible to run multiple CPUs per node using MPICH, but the MPICH must be built to allow it. Be aware that certain MPICH applications may not work perfectly in a threaded environment (see the Known Limitations section in the *Release Notes*). The safest thing for multiple CPUs, although not necessarily the fastest, is to run one MPI process per processor with `OMP_NUM_THREADS` set to one. Always verify that the combination with `OMP_NUM_THREADS=1` works correctly.

Using Shared Libraries

All needed shared libraries must be visible on all the nodes at run time. One way to accomplish this is to point these libraries by the `LD_LIBRARY_PATH` environment variable in the `.bashrc` file. If Intel MKL is installed only on one node, you should link statically when building your Intel MKL applications.

The Intel® compilers or GNU compilers can be used to compile a program that uses Intel MKL. However, make certain that MPI implementation and compiler match up correctly.

ScaLAPACK Tests

To build NetLib ScaLAPACK tests for IA-32, IA-64, or Intel® 64 architectures, add `libmkl_scalapack_core.a` to your link command.

Examples for Linking with ScaLAPACK and Cluster FFT

For information on detailed MKL structure of the architecture-specific directories of the cluster libraries, see section Directory Structure in Detail in Chapter 3.

Examples for C Module

Suppose the following conditions are met:

- MPICH 1.2.5 or higher is installed in `/opt/mpich`,
- Intel® MKL 10.0 is installed in `/opt/intel/mkl/10.0.xxx`, where `xxx` is the Intel MKL package number, for example, `/opt/intel/mkl/10.0.039`.
- You use Intel® C Compiler 8.1 or higher and the main module is in C.

To link with ScaLAPACK for a cluster of systems based on the IA-32 architecture, use the following libraries:

```
/opt/mpich/bin/mpicc <user files to link>     \
     -L/opt/intel/mkl/10.0.xxx/lib/32         \
     -lmkl_scalapack_core                     \
     -lmkl_blacs                              \
     -lmkl_lapack                             \
     -lmkl_intel -lmkl_intel_thread -lmkl_core \
     -lguide                                  \
     -lpthread
```

To link with Cluster FFT for a cluster of systems based on the IA-64 architecture, use the following libraries:

```
/opt/mpich/bin/mpicc <user files to link>     \
     -L/opt/intel/mkl/10.0.xxx/lib/64         \
     -lmkl_cdft_core                          \
     -lmkl_blacs_ilp64                        \
     -lmkl_intel -lmkl_intel_thread -lmkl_core \
     -lguide -lpthread
```

Examples for Fortran Module

Suppose the following conditions are met:

- Intel MPI 3.0 is installed in `/opt/intel/mpi/3.0`,
- Intel® MKL 10.0 is installed in `/opt/intel/mkl/10.0.xxx`, where `xxx` is the Intel MKL package number, for example, `/opt/intel/mkl/10.0.039`.
- You use Intel® Fortran Compiler 8.1 or higher and the main module is in Fortran.

To link with ScaLAPACK for a cluster of systems based on the IA-64 architecture, use the following libraries:

```
/opt/intel/mpi/3.0/bin/mpiifort <user files to link>       \
     -L/opt/intel/mkl/10.0.xxx/lib/64                      \
     -lmkl_scalapack_lp64                                  \
     -lmkl_blacs_intelmpi20_lp64                           \
     -lmkl_lapack                                          \
     -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core         \
     -lguide                                               \
     -lpthread
```

To link with Cluster FFT for a cluster of systems based on the IA-64 architecture, use the following libraries:

```
/opt/intel_mpi_10/bin/mpiifort <user files to link> \
     -L/opt/intel/mkl/10.0.xxx/lib/64               \
     -lmkl_cdft_core                                \
     -lmkl_blacs_intelmpi_ilp64                     \
     -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core  \
     -lguide -lpthread
```

A binary linked with ScaLAPACK runs in the same way as any other MPI application (For information, refer to the documentation that comes with the MPI implementation). For instance, the script `mpirun` is used in case of MPICH 1.2.x and OpenMPI, and a number of MPI processes is set by `-np`. In case of MPICH 2.0 and all Intel MPIs, you should start the daemon before running an application; the execution is driven by the script `mpiexec`.

For further linking examples, see the Intel MKL support website at http://www.intel.com/support/performancetools/libraries/mkl/.

LINPACK and MP LINPACK Benchmarks

10

This chapter describes the Intel® Optimized LINPACK Benchmark for Linux* and Intel® Optimized MP LINPACK Benchmark for Clusters.

Intel® Optimized LINPACK Benchmark for Linux*

Intel® Optimized LINPACK Benchmark is a generalization of the LINPACK 1000 benchmark. It solves a dense (`real`*8) system of linear equations (*Ax=b*), measures the amount of time it takes to factor and solve the system, converts that time into a performance rate, and tests the results for accuracy. The generalization is in the number of equations (*N*) it can solve, which is not limited to 1000. It uses partial pivoting to assure the accuracy of the results.

This benchmark should not be used to report LINPACK 100 performance, as that is a compiled-code only benchmark. This is a shared memory (SMP) implementation which runs on a single platform and should not be confused with MP LINPACK, which is a distributed memory version of the same benchmark. This benchmark should not be confused with LINPACK, the library, which has been expanded upon by the LAPACK library.

Intel is providing optimized versions of the LINPACK benchmarks to make it easier than using HPL for you to obtain high LINPACK benchmark results on your systems based on genuine Intel® processors. Use this package to benchmark your SMP machine.

Additional information on this software as well as other Intel® software performance products is available at http://developer.intel.com/software/products/.

Contents

The Intel Optimized LINPACK Benchmark for Linux* contains the following files, located in the `./benchmarks/linpack/` subdirectory in the Intel MKL directory (see Table 3-1):

Table 10-1 Contents of the LINPACK Benchmark

./benchmarks/linpack/	
linpack_itanium	The 64-bit program executable for a system based on Intel® Itanium® 2 processor.
linpack_xeon32	The 32-bit program executable for a system based on Intel® Xeon® processor or Intel® Xeon® processor MP with or without Streaming SIMD Extensions 3 (SSE3).
linpack_xeon64	The 64-bit program executable for a system with Intel® Xeon® processor using Intel® 64 architecture.
runme_itanium	A sample shell script for executing a pre-determined problem set for linpack_itanium. OMP_NUM_THREADS set to 8 processors.
runme_xeon32	A sample shell script for executing a pre-determined problem set for linpack_xeon32. OMP_NUM_THREADS set to 2 processors.
runme_xeon64	A sample shell script for executing a pre-determined problem set for linpack_xeon64. OMP_NUM_THREADS set to 4 processors.
lininput_itanium	Input file for pre-determined problem for the runme_itanium script.
lininput_xeon32	Input file for pre-determined problem for the runme_xeon32 script.
lininput_xeon64	Input file for pre-determined problem for the runme_xeon64 script.
lin_itanium.txt	Result of the runme_itanium script execution.
lin_xeon32.txt	Result of the runme_xeon32 script execution.
lin_xeon64.txt	Result of the runme_xeon64 script execution.
help.lpk	Simple help file.
xhelp.lpk	Extended help file.

Running the Software

To obtain results for the pre-determined sample problem sizes on a given system, type one of the following, as appropriate:

```
./runme_itanium
./runme_xeon32
./runme_xeon64 .
```

To run the software for other problem sizes, please refer to the extended help included with the program. Extended help can be viewed by running the program executable with the "-e" option:

```
./xlinpack_itanium -e

./xlinpack_xeon32 -e

./xlinpack_xeon64 -e .
```

The pre-defined data input files `lininput_itanium`, `lininput_xeon32`, and `lininput_xeon64` are provided merely as examples. Different systems may have different number of processors, or amount of memory, and require new input files. The extended help can be used for insight into proper ways to change the sample input files.

Each input file requires at least the following amount of memory:

`lininput_itanium`	16 GB
`lininput_xeon32`	2 GB
`lininput_xeon64`	16 GB.

If the system has less memory than the above sample data inputs require, you may have to edit or create your own data input files, as directed in the extended help.

Each sample script, in particular, uses the `OMP_NUM_THREADS` environment variable to set the number of processors it is targeting. To optimize performance on a different number of physical processors, change that line appropriately. If you run the Intel Optimized LINPACK Benchmark without setting the number of threads, it will default to the number of cores according to the OS. You can find the settings for this environment variable in the `runme_*` sample scripts. If the settings do not already match the situation for your machine, edit the script.

Known Limitations

The following limitations are known for the Intel Optimized LINPACK Benchmark for Linux*:

- Intel Optimized LINPACK Benchmark is threaded to effectively use multiple processors. So, in multi-processor systems, best performance will be obtained with Hyper-Threading technology turned off, which ensures that the operating system assigns threads to physical processors only.
- If an incomplete data input file is given, the binaries may either hang or fault. See the sample data input files and/or the extended help for insight into creating a correct data input file.

Intel® Optimized MP LINPACK Benchmark for Clusters

The Intel® Optimized MP LINPACK Benchmark for Clusters is based on modifications and additions to HPL 1.0a from Innovative Computing Laboratories (ICL) at the University of Tennessee, Knoxville (UTK). The benchmark can be used for Top 500 runs (see http://www.top500.org). The use of the benchmark requires that you are already intimately familiar with the HPL distribution and usage. This package adds some additional enhancements and bug fixes designed to make the HPL usage more convenient. The `benchmarks/mp_linpack` directory adds techniques to minimize search times frequently associated with long runs.

The Intel® Optimized MP LINPACK Benchmark for Clusters is an implementation of the Massively Parallel MP LINPACK benchmark. HPL code was used as a basis. It solves a random dense (`real`*8) system of linear equations (*Ax=b*), measures the amount of time it takes to factor and solve the system, converts that time into a performance rate and tests the results for accuracy. You can solve any size (*N*) system of equations that fit into memory. The benchmark uses full row pivoting to ensure the accuracy of the results.

This benchmark should not be used to report LINPACK performance on a shared memory machine. For that, the Intel® Optimized LINPACK Benchmark should be used instead. This benchmark should be used on a distributed memory machine.

Intel is providing optimized versions of the LINPACK benchmarks to make it easier than using HPL for you to obtain high LINPACK benchmark results on your systems based on genuine Intel® processors. Use this package to benchmark your cluster. The prebuilt binaries require Intel® MPI 3.x be installed on the cluster. The run-time version of Intel MPI is free and can be downloaded from www.intel.com/software/products/cluster.

NOTE. If you wish to use a different version of MPI, you can do so by using the MP LINPACK source provided.

The package includes software developed at the University of Tennessee, Knoxville, Innovative Computing Laboratories and neither the University nor ICL endorse or promote this product. Although HPL 1.0a is redistributable under certain conditions, this particular package is subject to the MKL license.

Contents

The Intel Optimized MP LINPACK Benchmark for Clusters includes the HPL 1.0a distribution in its entirety as well as the modifications, delivered in the files listed in Table 10-2 and located in the `/benchmarks/mp_linpack/` subdirectory in the Intel MKL directory (see Table 3-1):

Table 10-2 Contents of the MP LINPACK Benchmark

./benchmarks/mp_linpack/	
testing/ptest/HPL_pdtest.c	HPL 1.0a code modified to display captured DGEMM information in ASYOUGO2_DISPLAY (see details in the New Features section) if it was captured.
src/blas/HPL_dgemm.c	HPL 1.0a code modified to capture DGEMM information if desired from ASYOUGO2_DISPLAY
src/grid/HPL_grid_init.c	HPL 1.0a code modified to do additional grid experiments originally not in HPL 1.0.
src/pgesv/HPL_pdgesvK2.c	HPL 1.0a code modified to do ASYOUGO and ENDEARLY modifications
include/hpl_misc.h and hpl_pgesv.h	Bugfix added to allow for 64-bit address computation.
src/pgesv/HPL_pdgesv0.c	HPL 1.0a code modified to do ASYOUGO, ASYOUGO2, and ENDEARLY modifications
testing/ptest/HPL.dat	HPL 1.0a sample HPL.dat modified.
Make.ia32	(New) Sample architecture make for processors using IA-32 architecture and Linux.
Make.em64t	(New) Sample architecture make for processors using Intel® 64 architecture and Linux.
Make.ipf	(New) Sample architecture make for IA-64 architecture and Linux.
Next three files are prebuilt executables, readily available for simple performance testing.	
bin_intel/ia32/xhpl_ia32	(New) Prebuilt binary for IA-32 architecture, Linux, and Intel® MPI 3.0.
bin_intel/em64t/xhpl_em64t	(New) Prebuilt binary for Intel® 64 architecture, Linux, and Intel MPI 3.0.
bin_intel/ipf/xhpl_ipf	(New) Prebuilt binary for IA-64 architecture, Linux, and Intel MPI 3.0.
HPL.dat	A repeat of testing/ptest/HPL.dat in the top-level directory
nodeperf.c	(New) Sample utility that tests the DGEMM speed across the cluster.

Building MP LINPACK

There are a few included sample architecture makes. It is recommended that you edit them to fit your specific configuration. In particular:

- Set `TOPdir` to the directory MP LINPACK is being built in.
- You may set MPI variables, that is, `MPdir`, `MPinc`, and `MPlib`.
- Specify the location of Intel MKL and of files to be used (`LAdir`, `LAinc`, `LAlib`).
- Adjust compiler and compiler/linker options.

For some sample cases, like Linux systems based on Intel® 64 architecture, the makes contain values that seem to be common. However, you are required to be familiar with building HPL and picking appropriate values for these variables.

New Features

The toolset is basically identical with the HPL 1.0a distribution. There are a few changes which are optionally compiled in and are disabled until you specifically request them. These new features are:

ASYOUGO: Provides non-intrusive performance information while runs proceed. There are only a few outputs and this information does not impact performance. This is especially useful because many runs can go hours without any information.

ASYOUGO2: Provides slightly intrusive additional performance information because it intercepts every `DGEMM`.

ASYOUGO2_DISPLAY: Displays the performance of all the significant `DGEMMs` inside the run.

ENDEARLY: Displays a few performance hints and then terminates the run early.

FASTSWAP: Inserts the LAPACK-optimized `DLASWP` into HPL's code. This may yield a benefit for Itanium® 2 processor. You can experiment with this to determine best results.

Benchmarking a Cluster

To benchmark a cluster, follow the sequence of steps (maybe, optional) below. Pay special attention to the iterative steps 3 and 4. They make up a loop that searches for HPL parameters (specified in `HPL.dat`) which the top performance of you cluster is reached with.

1. Get HPL installed and functional on all the nodes.
2. You may run `nodeperf.c` (included in the distribution) to see the performance of `DGEMM` on all the nodes.

   Compile `nodeperf.c` in with your MPI and Intel MKL.

   For example,

   ```
   mpicc -O3 nodeperf.c /opt/intel/mkl/10.0.xxx/lib/em64t/libmkl_em64t.a
   /opt/intel/mkl/10.0.xxx/lib/em64t/libguide.a -lpthread -o nodeperf
   ```

   where `xxx` is the Intel MKL package number.

   Launching `nodeperf.c` on all the nodes is especially helpful in a very large cluster. Indeed, there may be a stray job on a certain node, for example, 738, which is running 5% slower than the rest. MP LINPACK will then run as slow as the slowest node. In this case, `nodeperf` enables quick identifying of the potential problem spot without lots of small MP LINPACK runs around the cluster in search of the bad node. It is common that after a bunch of HPL runs, there may be zombie processes and `nodeperf` facilitates finding the slow nodes. It goes through all the nodes, one at a time, and reports the performance of `DGEMM` followed by some host identifier. Therefore, the higher the penultimate number then, the faster that node was performing.
3. Edit `HPL.dat` to fit your cluster needs.

   Read through the HPL documentation for ideas on this. However, you should try on at least 4 nodes.
4. Make an HPL run, using compile options such as `ASYOUGO` or `ASYOUGO2` or `ENDEARLY` to aid in your search (These options enable you to gain insight into the performance sooner than HPL would normally give this insight.)

   When doing so, follow these recommendations:

   — Use the MP LINPACK patched version of HPL to save time in the searching.

   Using a patched version of HPL should not hinder your performance. That's why features that could be performance intrusive are compile-optional (and it is called out below) in MP LINPACK. That is, if you don't use any of the new options explained in section Options to reduce search time, then these changes are disabled. The primary purpose of the additions is to assist you in finding solutions.

   HPL requires long time to search for many different parameters. In the MP LINPACK, the goal is to get the best possible number.

   Given that the input is not fixed, there is a large parameter space you must search over. In fact, an exhaustive search of all possible inputs is improbably large even for a powerful cluster.

   This patched version of HPL optionally prints information on performance as it proceeds, or even terminates early depending on your desires.

— Save time by compiling with `-DENDEARLY -DASYOUGO2` (described in the Options to reduce search time section) and using a negative threshold (Do not to use a negative threshold on the final run that you intend to submit if you are doing a Top500 entry!) You can set the threshold in line 13 of the HPL 1.0a input file `HPL.dat`.

— If you are going to run a problem to completion, do it with `-DASYOUGO` (see Options to reduce search time section).

5. Using the quick performance feedback, return to step 3 and iterate until you are sure that the performance is as good as possible.

Options to reduce search time

Running huge problems to completion on large numbers of nodes can take many hours. The search space for MP LINPACK is also huge: not only can you run any size problem, but over a number of block sizes, grid layouts, lookahead steps, using different factorization methods, etc. It can be a large waste of time to run a huge problem to completion only to discover it ran 0.01% slower than your previous best problem.

There are 3 options you might want to experiment with to reduce the search time:

- `-DASYOUGO`
- `-DENDEARLY`
- `-DASYOUGO2`

  Use cautiously, as it does have a marginal performance impact. To see `DGEMM` internal performance, compile with `-DASYOUGO2` and `-DASYOUGO2_DISPLAY`. This will give lots of useful `DGEMM` performance information at the cost of around 0.2% performance loss.

If you want the old HPL back, simply don't define these options and recompile from scratch (try "`make arch=<arch> clean all`").

-DASYOUGO: Gives performance data as the run proceeds. The performance always starts off higher and then drops because this actually happens in LU decomposition. The `ASYOUGO` performance estimate is usually an overestimate (because LU slows down as it goes), but it gets more accurate as the problem proceeds. The greater the lookahead step, the less accurate the first number may be. `ASYOUGO` tries to estimate where one is in the LU decomposition that MP LINPACK performs and this is always an overestimate as compared to `ASYOUGO2`, which measures actually achieved `DGEMM` performance. Note that the `ASYOUGO` output is a subset of the information that `ASYOUGO2` provides. So, refer to the description of the `-DASYOUGO2` option below for the details of the output.

-DENDEARLY: Terminates the problem after a few steps, so that you can set up 10 or 20 HPL runs without monitoring them, see how they all do, and then only run the fastest ones to completion. `-DENDEARLY` assumes `-DASYOUGO`. You do not need to define both, although it doesn't hurt. Because the problem terminates early, it is recommended setting the

"threshold" parameter in HPL.dat to a negative number when testing ENDEARLY. There is no point in doing a residual check if the problem ended early. It also sometimes gives a better picture to compile with -DASYOUGO2 when using -DENDEARLY.

You need to know the specifics of -DENDEARLY:

— -DENDEARLY stops the problem after a few iterations of DGEMM on the blocksize (the bigger the blocksize, the further it gets). It prints only 5 or 6 "updates", whereas -DASYOUGO prints about 46 or so outputs before the problem completes.

— Performance for -DASYOUGO and -DENDEARLY always starts off at one speed, slowly increases, and then slows down toward the end (because that is what LU does). -DENDEARLY is likely to terminate before it starts to slow down.

— -DENDEARLY terminates the problem early with an HPL Error exit. It means that you need to ignore the missing residual results, which are wrong, as the problem never completed. However, you can get an idea what the initial performance was, and if it looks good, then run the problem to completion without -DENDEARLY. To avoid the error check, you can set HPL's threshold parameter in HPL.dat to a negative number.

— Though -DENDEARLY terminates early, HPL treats the problem as completed and computes Gflop rating as though the problem ran to completion. Ignore this erroneously high rating.

— The bigger the problem, the more accurately the last update that -DENDEARLY returns will be close to what happens when the problem runs to completion. -DENDEARLY is a poor approximation for small problems. It is for this reason that you are suggested to use ENDEARLY in conjunction with ASYOUGO2, because ASYOUGO2 reports actual DGEMM performance, which can be a closer approximation to problems just starting.

The best known compile options for Itanium® 2 processor are with the Intel® compiler and look like this:

```
-O2 -ipo -ipo_obj -ftz -IPF_fltacc -IPF_fma -unroll -w -tpp2
```

-DASYOUGO2: Gives detailed single-node DGEMM performance information. It captures all DGEMM calls (if you use Fortran BLAS) and records their data. Because of this, the routine has a marginal intrusive overhead. Unlike -DASYOUGO, which is quite non-intrusive, -DASYOUGO2 is interrupting every DGEMM call to monitor its performance. You should beware of this overhead, although for big problems, it is, for sure, less than 1/10th of a percent.

Here is a sample ASYOUGO2 output (the first 3 non-intrusive numbers can be found in ASYOUGO and ENDEARLY), so it suffices to describe these numbers here:

```
Col=001280 Fract=0.050 Mflops=42454.99 (DT= 9.5 DF= 34.1
DMF=38322.78).
```

The problem size was *N*=16000 with a blocksize of 128. After 10 blocks, that is, 1280 columns, an output was sent to the screen. Here, the fraction of columns completed is 1280/16000=0.08. Only about 20 outputs are printed, at various places through the

matrix decomposition: `fractions 0.005,0.010,0.015,0.02,0.025,0.03,0.035,0.04,0.045,0.05,0.055,0.06,0.065,0.07,0.075,0.080,0.085,0.09,0.095,.10,...,.195,.295,.395,...,.895`. However, this problem size is so small and the block size so big by comparison that as soon as it printed the value for `0.045`, it was already through 0.08 fraction of the columns. On a really big problem, the fractional number will be more accurate. It never prints more than the 46 numbers above. So, smaller problems will have fewer than 46 updates, and the biggest problems will have precisely 46 updates.

The `Mflops` is an estimate based on 1280 columns of LU being completed. However, with lookahead steps, sometimes that work is not actually completed when the output is made. Nevertheless, this is a good estimate for comparing identical runs.

The 3 numbers in parenthesis are intrusive `ASYOUGO2` addins. The `DT` is the total time processor 0 has spent in `DGEMM`. The `DF` is the number of billion operations that have been performed in `DGEMM` by one processor. Hence, the performance of processor 0 (in Gflops) in `DGEMM` is always `DF/DT`. Using the number of `DGEMM` flops as a basis instead of the number of LU flops, you get a lower bound on performance of our run by looking at `DMF`, which can be compared to `Mflops` above (It uses the global LU time, but the `DGEMM` flops are computed under the assumption that the problem is evenly distributed amongst the nodes, as only HPL's node (0,0) returns any output.)

Note that when using the above performance monitoring tools to compare different `HPL.dat` inputs, you should beware that the pattern of performance drop off that LU experiences is sensitive to some of the inputs. For instance, when you try very small problems, the performance drop off from the initial values to end values is very rapid. The larger the problem, the less the drop off, and it is probably safe to use the first few performance values to estimate the difference between a problem size 700000 and 701000, for instance. Another factor that influences the performance drop off is the grid dimensions (P and Q). For big problems, the performance tends to fall off less from the first few steps when P and Q are roughly equal in value. You can make use of a large number of parameters, such as broadcast types, and change them so that the final performance is determined very closely by the first few steps.

Using these tools will greatly assist the amount of data you can test.

Intel® Math Kernel Library Language Interfaces Support

A

The following table shows language interfaces that Intel® Math Kernel Library (Intel® MKL) provides for each function domain. However, Intel MKL routines can be called from other languages using mixed-language programming. For example, see section "Mixed-language programming with Intel® MKL" in chapter 7 on how to call Fortran routines from C/C++.

Table A-1 Intel MKL language interfaces support

Function Domain	Fortran 77 interface	Fortran 90/95 interface	C/C++ interface
Basic Linear Algebra Subprograms (BLAS)	+	+	via CBLAS
Sparse BLAS Level 1	+	+	via CBLAS
Sparse BLAS Level 2 and 3	+	+	+
LAPACK routines for solving systems of linear equations	+	+	
LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations	+	+	
Auxiliary and utility LAPACK routines	+		
ScaLAPACK routines	+		
PARDISO	+		+
Other Direct and Iterative Sparse Solver routines	+	+	+
Vector Mathematical Library (VML) functions		+	+
Vector Statistical Library (VSL) functions		+	+
Fourier Transform functions (FFT)		+	+
Cluster FFT functions		+	+
Interval Solver routines	+		
Trigonometric Transform routines		+	+

Table A-1 Intel MKL language interfaces support

Function Domain	Fortran 77 interface	Fortran 90/95 interface	C/C++ interface
Fast Poisson, Laplace, and Helmholtz Solver (Poisson Library) routines		+	+
Optimization (Trust-Region) Solver routines	+	+	+

Support for Third-Party Interfaces

B

This appendix describes in brief certain interfaces that Intel® Math Kernel Library (Intel® MKL) supports.

GMP* Functions

Intel MKL implementation of GMP* arithmetic functions includes arbitrary precision arithmetic operations on integer numbers. The interfaces of such functions fully match the GNU Multiple Precision* (GMP) Arithmetic Library.

If you currently use the GMP* library, you need to modify `INCLUDE` statements in your programs to `mkl_gmp.h`.

FFTW Interface Support

Intel MKL offers two wrappers collections, each being the FFTW interface superstructure, to be used for calling the Intel MKL Fourier transform functions. These collections correspond to the FFTW versions 2.x and 3.x, respectively, and the Intel MKL versions 7.0 and later.

The purpose of these wrappers is to enable developers whose programs currently use FFTW to gain performance with the Intel MKL Fourier transforms without changing the program source code. See *FFTW to Intel® MKL Wrappers Technical User Notes for FFTW 2.x* (`fftw2xmkl_notes.htm`) for details on the use of the FFTW 2.x wrappers and *FFTW to Intel® MKL Wrappers Technical User Notes for FFTW 3.x* (`fftw3xmkl_notes.htm`) for details on the use of the FFTW 3.x wrappers.

Index

A

affinity mask, 6-14
audience, 1-2

B

benchmark, 10-1
BLAS
  calling routines from C, 7-5
  Fortran 95 interfaces to, 7-2

C

C, calling LAPACK, BLAS, CBLAS from, 7-4
calling
  BLAS functions in C, 7-6
  complex BLAS Level 1 function from C, 7-7
  complex BLAS Level 1 function from C++, 7-8
  Fortran-style routines from C, 7-4
CBLAS, 7-6
CBLAS, code example, 7-9
Cluster FFT, linking with, 9-1
cluster software, 9-1
  linking examples, 9-3
  linking syntax, 9-1
coding
  data alignment, 8-1
  mixed-language calls, 7-6
  techniques to improve performance, 6-12
Compatibility OpenMP run-time compiler library, 5-7
compiler support, 2-2
compiler support RTL layer, 3-4
compiler-dependent function, 7-3
computational layer, 3-4
configuration file, 4-4
configuring development environment, 4-1
  Eclipse CDT, 4-2
  redefining library names, 4-5
custom shared object, 5-11
  building, 5-11
  specifying list of functions, 5-12
  specifying makefile parameters, 5-11

D

denormal, performance, 6-15
development environment, configuring, 4-1
directory structure
  documentation, 3-19
  high-level, 3-1
  in-detail, 3-11
documentation, 3-19
dummy library, 3-19
dynamic linking, 5-2

E

Eclipse CDT, configuring, 4-2
environment variables, setting, 4-1
examples, ScaLAPACK, Cluster FFT, linking with, 9-3

F

FFT functions, data alignment, 6-13
FFT interface
  MKL_LONG type, 3-6
  optimized radices, 6-15
  threading tip, 6-12
FFTW interface support, B-1
Fortran 95, interfaces to LAPACK and BLAS, 7-2

G

GMP arithmetic functions, B-1
GNU Multiple Precision Arithmetic Library, B-1

H

HT Technology, *see* Hyper-Threading technology
Hyper-Threading Technology, configuration tip, 6-13

I

ILP64 programming, support for, 3-5
installation, checking, 2-1
interface layer, 3-4

J

Java examples, 7-10

L

language interfaces support, A-1
  Fortran 95 interfaces, 7-2
  language-specific interfaces, 7-1
LAPACK
  calling routines from C, 7-4
  Fortran 95 interfaces to, 7-2
  packed routines performance, 6-12
layer
  compiler support RTL, 3-4
  computational, 3-4
  interface, 3-4
  RTL, 3-3
  threading, 3-4
layered model, 3-2
Legacy OpenMP run-time compiler library, 5-7
library
  run-time, Compatibility OpenMP, 5-7
  run-time, Legacy OpenMP, 5-7
library names, redefining in config file, 4-5
library structure, 3-1
link command
  examples, 5-8
  syntax, 5-3
link libraries, 5-6
linkage models, comparison, 5-2
linking, 5-1
  default model, 5-4
  dynamic, 5-2
  pure layered model, 5-4
  recommendations, 5-3
  static, 5-1
  with Cluster FFT, 9-1
  with ScaLAPACK, 9-1
LINPACK benchmark, 10-1

M

memory functions, redefining, 6-16
memory management, 6-15
memory renaming, 6-16
mixed-language programming, 7-4
module, Fortran 95, 7-4
MP LINPACK benchmark, 10-4
multi-core performance, 6-14

N

notational conventions, 1-3
number of threads
  changing at run time, 6-4
  setting for cluster, 9-2
  setting with OpenMP environment variable, 6-4
  techniques to set, 6-2
numerical stability, 8-1

O

OpenMP
  Compatibility run-time compiler library, 5-7

Legacy run-time compiler library, 5-7

P

parallel performance, 6-3
parallelism, 6-1
performance, 6-1
coding techniques to gain, 6-12
hardware tips to gain, 6-13
multi-core, 6-14
of LAPACK packed routines, 6-12
with denormals, 6-15

R

RTL, 7-3
RTL layer, 3-3
run-time library, 7-3
Compatibility OpenMP, 5-7
Legacy OpenMP, 5-7

S

ScaLAPACK, linking with, 9-1
sequential version of the library, 3-4
static linking, 5-1
support, technical, 1-1
syntax
linking, cluster software, 9-1
linking, general, 5-3

T

technical support, 1-1
threading
avoiding conflicts, 6-3
environment variables and functions, 6-7
Intel MKL controls, 6-7
see also number of threads
threading layer, 3-4

U

usage information, 1-1